PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT.

COMMITTED CHOICE FUNCTIONAL PROGRAMMING

Göran Båge

Computer Science Laboratory
Ellemtel
S-125 25 Älvsjö, Sweden

Gary Lindstrom

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

ABSTRACT

We outline the design, semantics, implementation, and application of a normal order functional language enriched with logical variables and committed choice indeterminacy. Our notion of committed choice functional programming, motivated by Ueda's Guarded Horn Clauses, is based on trial unification of formal parameters (serving as guards) and actual parameters (serving as goals). An implementation based on fine grain (S, K, I) combinators is described. Compilation is accomplished through a simple extension of Turner's classical abstraction method, accompanied by a parallel combinator graph reducer. This system uses three extensions in support of committed choice: (i) a guard notation applicable at each level of function Currying; (ii) a `solve` library function that conducts commitment competitions, and (iii) a combinator `R` that "protects" logical variables in goals during trial unification (unification where all goal variables are treated as read-only). The reducer has been implemented and empirically validated on machines with both pseudo and real concurrency. Pragmatically, we have learned that normal order and unification are a very powerful semantic combination, but require careful use due to the semantic visibility of subtle evaluation causality effects. These effects are illustrated on a one-pass definition/use example using indeterminate fair merge.

1 FUNCTIONAL PROGRAMMING WITH LOGICAL VARIABLES

1.1 The Added Power of Logical Variables

Functional and logic programming languages share common roots as declarative programming methodologies, but exhibit several distinguishing characteristics as well. Functional programming relies heavily on algebraic concepts, with a strong emphasis on combining forms, facilitated by higher order functionals and Curried notation. However, the pervasive producer-consumer orientation of functional programming dictates that all variables must be bound at the place of their introduction.

In contrast, logic programming offers the notions of *logical variables* and delayed binding via *unification*. Several researchers have discovered that logical variables, even when unification is required to always succeed, can add significant power to normal order functional programming languages. Capabilities facilitated by this extension include:

1. Computation by constraint intersection, e.g. polymorphic type checking (Milner 1978);
2. "Micro object" support for object oriented programming, e.g. for direct message delivery;
3. Monotonic refinement of shared data structures, e.g. functional representation of graph reduction interpreters (Pingali 1986, 1987), and
4. Stream interconnection of processes that are not designated *a priori* as producers or consumers, e.g. simulation of hardware systems involving bidirectional or broadcast communication (buses or pass transistors).

1.2 Logical Variables and Functional Processes

It has been widely observed that normal order functional languages (NOFL's) constitute a natural framework for diffusing processes over parallel architectures. These process relationships, simple or intricate, fundamentally rely on overlapped production and consumption of lazily evaluated data structures (notably streams). All efficient implementations of functional programming exploit this predictable information flow, e.g. dataflow, demand propagation, or normal order graph reduction.

When logical variables are introduced into a functional language, the induced process organization becomes more subtle. In effect, logical variables permit the deferred specification of the "producer" of the binding of a variable, resulting in unpredictable information flow directionality among processes (Reddy 1986). As Shapiro has observed (Shapiro 1983), this "isotropic" flow of information through logical variables greatly enriches the possible patterns of process synchronization and communication.

One may ask whether the traditional NOFL implementation strategies can be adapted to deal with these new dimensions of process interaction, while retaining lazy semantics, higher order functions, and Currying. In a previous paper (Båge and Lindstrom 1987) we showed that the answer is affirmative, at least for S, K, I combinator reduction implementations.

Nevertheless, the semantic and pragmatic interactions of normal order evaluation and unification can be quite subtle. Introducing logical variables weakens the doctrine of referential transparency, by the following argument. Suppose a function `lv(n)` delivers a list of n logical variables. Then whether `append(lv(n), lv(n))` shares or replicates its calls on `lv(n)` will be semantically crucial (e.g. suppose the appended list is unified with `[1, 2, ..., 2*n]`).

Moreover, since the "read-only" access of a logical variable does not "force" its evaluation, processes making such accesses must be prepared to suspend, awaiting a future "action at a distance" binding the variable. The resulting looser process coupling has many advantages, but poses new implementation challenges as well.

1.3 Dealing With Unification Failure

In (Lindstrom 1985) it is shown that logical variables can be incorporated into a NOFL while preserving parallelism, determinacy, and "definiteness". By the last term we mean, informally, "each complete response to a finite top-level demand pattern, not reporting a unification error, will not later be determined to involve a unification error unless the demand pattern is refined."

While this well-behavedness for "correct" programs is pleasing, we are nevertheless left with a vexing problem: how to recover from unification failures. Note that in a parallel setting an `if_unify(x, y)` predicate would be necessarily be indeterminate since its result may be `true` or `false` depending on the current state of evaluation of `x` and `y`, if their ultimate bindings are incompatible.

Concurrent logic programming languages must deal with the same phenomenon. While some researchers have proposed to retain (at least set-level) determinacy by careful blending of AND and OR parallelism, others have chosen to embrace indeterminacy and exploit its operational sensibility and applicational versatility. In this paper we follow in the footsteps of the latter "committed choice" language designers, and adapt their concepts to a NOFL with logical variables.

2 SASL+LV

In (Båge and Lindstrom 1987) we introduce SASL+LV, a prototypical NOFL with logical variables.[1] This language is meant to be a platform for experimentation in semantic "gene splicing," and not an example of how logical variables should be introduced in a fully reasoned manner into a lazy functional language.

SASL+LV is (essentially) a subset of SASL (Turner 1983), extended to include logical variables and parameter passage by unification. SASL+LV differs syntactically from its base language 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 the atom `ERROR`.

2.1 Semantic Impact

The impact of logical variables on the formal semantics of SASL+LV is essentially the injection of a well-

[1] The name is in homage to David Turner, and is not intended to suggest that SASL+LV bears any direct relationship to the "real" SASL, or its descendants.

```
diff (diff((1:(2:X)):X) ((3:Y):Y))
     ((4:nil):nil)
     where
          diff (A:B) (B:C) = A:C
```

Figure 1: Sample SASL+LV function.

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.

To illustrate, consider the SASL+LV function

```
A:B B:C = A:C
```

which mimics the Prolog append relation on a form of "difference lists". We assume that `A`, `B`, and `C` all denote logical variables (we adopt the upper case initial letter convention of Prolog). The right associative infix operator `:` denotes lazy pairing. The application `diff (1:2:X):X (3:nil):nil` (where `nil` is a special list ending atom) returns `(1:2:3:nil):nil`. The correctness of `diff` directly depends on both occurrences of symbol `B` denoting *the same* logical variable.

A combinator-based implementation strategy for SASL+LV is described in detail in (Båge and Lindstrom 1987). As a foundation for our committed choice extension, we briefly summarize its salient features here.

2.2 FN Compilation

The general rule for compiling a function

```
f formal = body
```

is as follows. Suppose `formal` introduces n logical variables, $v_1, \ldots, v_n$. 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` (`P` is the lazy pairing combinator). 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 $v_1, \ldots, v_n$ are all distinct, conventional Turner compilation applies, i.e. `g' = [v1](...([vn](P formal body)) ...)`.

```
[Rule RFN0:]

  FN (P formal body) actual =>
      COND (UNIFY formal actual) body ERROR


[Rule RFN1:]

  FN fb actual =>
      FN (fb new_lv) actual
```

Figure 2: `FN` combinator reduction rules.

  - As in rule `CFN0`, inner function definitions must be compiled in cognizance of rules `CFN0` and `CFN1`.

2.3 FN Graph Reduction

The reduction rules for the `FN` combinator are given in Fig. 2. 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`.
4. The notation `new_lv` indicates the creation of a new logical variable node. Logical variables are represented as node sets tail-chained into cycles. A logical variable is created as a chain of length one, denoted `x->(LV x)`.

As described in item 4, logical variables are represented as circular chains of nodes, anchored by a unique instance of the `LV` combinator. Other logical variables equated to the base `LV` are chained in as `BV` combinators. Finally, read-only accesses of the `LV` anchoring the cycle are also chained in, pending notification of its binding.

The rules for `UNIFY` exploit this chained representation of logical variables. `UNIFY` is strict in both its arguments. Once both arguments are evaluated (to an atom, pair or logical variable), the `UNIFY` reduction rules illustrated in Fig. 3 are applied. Of special interest

(Highest priority)

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

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

[Rule UN3:]
   UNIFY x->(LV tailx)
         y              =>  I TRUE
                            (bind x to y)

[Rule UN5:]
   UNIFY x
         y->(LV taily)) =>  I TRUE
                            (bind y to x)

[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)

Figure 3: Unification reduction rules (partial list).

is rule UN2, which equates two logical variables. This action is accomplished by "twisting" their tail links, thereby merging their cycles, and converting one of their LV combinators to BV. (The potentially anomalous case of equating a variable to itself is handled by rule UN1.)

3 GUARDED SASL+LV

We now consider the primary subject of this paper: extending SASL+LV to encompass a form of committed choice indeterminacy.

3.1 Extensions to SASL+LV

The key ingredients in our committed choice extension are the following:

1. *Goals:* a means of applying competing guarded functions to a common actual parameter.
2. *Guarded functions:* syntactic and semantic devices that designate functions as being guarded in their application to goals. In the spirit of SASL, this should be done per Currying level.
3. *Trial unification:* insurance that logical variables in goals should not become bound during committed choice competitions.
4. *Arbitration and commitment:* an indeterminate operator that arbitrates among competing guarded functions, and selects an individual winner from among those reporting successful trial unification with the given goal.

3.2 Syntax and Semantics

Goal expressions require no special notation. Guarded functions are indicated by the keyword `guard` preceding the formal parameter to be unified at that Currying level. Committed choice competition is denoted by a `solve` operator, `solve` $[g_1, \ldots, g_k]$ `goal`, for $k > 0$.

The informal semantics of `solve` $[g_1, \ldots, g_k]$ `goal` is as follows:

1. An arbitrary selection is made of some g_i that trial unifies with `goal`.
2. If none trial unifies, `solve` returns `ERROR`.
3. Otherwise, `solve` applies g_i to `goal` using *true* unification.

We illustrate our experimental notation for Guarded SASL+LV on a "fair" merge pseudo-function, shown in Fig. 4.

```
merge [a, b] =
   solve [g1, g2, g3] [a, b]
   where
         g1 (guard [H : T, S]) =
                   H : merge [S, T],
         g2 (guard [S, H : T]) =
                   H : merge [T, S],
         g3 (guard [nil, nil]) =
                   nil
```

Figure 4: Fair merge in Guarded SASL+LV.

```
solve glist goal =
   glist = nil -> ERROR;
   gi goal
where
   gi:gtail = glist,
   gi = winner (g1 goal) (solve gtail goal),
   winner x y =
      commit x y -> x;
      y
```

Figure 5: The solve operation.

```
COMMIT x y           =>  TRUE  {x != ERROR}
COMMIT x y           =>  FALSE {y != ERROR}
COMMIT ERROR ERROR   =>  TRUE
COMMIT ERROR ERROR   =>  FALSE
```

Figure 6: Commit combinator definition.

4 COMPILATION OF GUARDED SASL+LV PROGRAMS

4.1 Guarded Function Compilation

The compilation schema in Section 2.2 can be adapted to deal with guarded functions by the introduction of one new combinator G. In fact, the compilation of a guarded function is simply that of its unguarded analog `f formal = body`, but with the G combinator substituted for the FN combinator: `(G [v1](...[vn](P formal body)...))`.

4.2 The solve Operation

Fig. 5 gives the definition of solve. The auxiliary function winner handles arbitration and commitment of guarded functions in glist, as applied to goal. The delivered guarded function (in *unguarded* form, indicated by a non-ERROR value) is then applied to goal. Note that solve itself does not use logical variables, let alone guards.

The commit operation translates to a direct application of the pseudo combinator COMMIT, defined by the indeterminate rules given in Fig. 6. Although problematical from a formal semantic standpoint, its operational semantics are clear and easy to implement.

```
[Rule RG0:]
      G fb actual => H fb fb actual

[Rule RH0:]
      H (P formal body) fb actual =>
         COND (UNIFY formal (R actual))
              (FN fb)
              ERROR)

[Rule RH1:]
      H fb fb' actual =>
         H (fb new_lv) fb' actual
```

Figure 7: G and H reduction rules.

5 PARALLEL GRAPH REDUCTION OF GUARDED SASL+LV

It remains for us to describe how guarded functions can be equipped to first do trial unification on a goal, and then deliver unguarded instances of themselves if that trial unification succeeds. To achieve this, we must introduce two additional combinators H and R.

5.1 G Combinator Reduction

The G combinator, and its partner H, serve to introduce read-only protection of the goal (actual parameter) during guard trial unification. This is arranged by reduction rules that are close variants of RFN0 and RFN1, shown in Fig. 7. Rule RH1 applies only when `fb` $\neq$ `(P a b)`. Note the double instance of fb in the righthand side of rule RG0. The first occurrence is instantiated with (local) logical variables by applications of rule RH1. When these instantiations are completed, trial unification is performed by rule RH0. The second instance of the original fb has been kept "pristine", and can be used to construct the unguarded instance of the function in `(FN fb)`.

5.2 Trial Unification In Guarded Functions

Rule RH0 performs trial unification of the function guard (formal parameter) and the goal (actual parameter) with the aid of the R combinator. The R combinator reduction rules are as follows:

```
[if a is not LV:]
   R (a b)     =>        (R a) (R b)

[if a is an atom:]
   R a         =>        I a
```

Note that an R combinator forms a barrier between the UNIFY that introduces it and any non-local logical variable imported by a goal. In particular, any function bodies referenced via R combinators have R-protection distributed throughout them, lest they be carelessly ignored as "Trojan Horses" bringing in unprotected external logical variables.

5.3 R Combinator Operational Control

The R rules can be freely applied. However, like all combinator reduction rules, they must be applied with a measure of restraint to avoid massive node copying or even needless divergence. In fact, the R rules can be applied very sparingly. The following "lazy" strategy is sufficiently thorough, and does not incur excessive R combinator propagation. To reduce (R x):

1. Evaluate x.
2. If x = (LV y) for some y, suspend unless a UNIFY rule applies (see Section 5.4).
3. If x = (a b c), a must be in normal form. Reduce (R x) to (a (R b) (R c)).
4. If x = (a b), again a must be in normal form. Reduce (R x) to (a (R b)).
5. Otherwise, x must be an atom; reduce (R x) to (I x).

Several desirable effects result:

- The evaluation of x in a (R x) expression is done in the ordinary (normal order) manner. This obviates any need for a special "read-only" evaluation mode.
- Guarded functions will be protected from unifying any LV's accessed through goals, as noted above.
- After evaluation of (R x), all R applications within x will be "pushed below" the levels inspected by any reduction rules applicable to this (or other) usage of x. Examples:

```
1. (hd (R (P a b)))
   => (hd (P (R a) (R b)))
   => (R a);
2. (R (S f g) y)
   => (S (R f) (R g) y)
   => ((R f y) (R g y)).
```

5.4 UNIFY Rules

We observe that by the operational policy just described, any evaluated argument (R z) to a UNIFY combinator can only be of the form (R (LV x)). The unification rules in Section 2.3 (notably UN3 and UN5) will handle all the resulting cases except the following:

1. (UNIFY (R (LV x)) y): Necessarily, y is not of the form (LV z), since that case is handled by rule UN5. Action: suspend until (LV x) becomes bound to a non-LV value.
2. (UNIFY x (R (LV y))): Symmetric to case 1.

Note that the case (UNIFY (R (LV x)) (R (LV y))) is subsumed by these rules. The effect in this case is to suspend on the first argument, and then on the second (since the UNIFY cannot complete until *both* become bound).

6 EMPIRICAL RESULTS

6.1 Implementation Status

Guarded SASL+LV has been implemented and validated on examples such as the merge-based one-pass symbol definition/usage problem given in Fig. 8 ((0:id) indicates *usage* of label id, and (1:id) indicates *definition* of label id). Our compiler generates a graph with 634 combinators for this example; reduction yields the following answer within a few seconds on a VAX 8600:

```
[0:7, 0:3, 1:3, 0:8, 0:8, 1:6, 1:7, 1:8]
```

Our compiler uses *yacc* to produce an abstract syntax tree, which is then converted to a combinator tree through a compiler written in approximately five pages of Standard ML. The reducer is written in C (slightly over 3000 source lines), and builds on the link permutation techniques used in SKIM (Stoye et al. 1984) and Norma (Scheevel 1986) . As explained earlier, the use of some form of multi-tasking (at least simulated) is inherent in the semantics of SASL+LV. We model such concurrency using a Simula-like simulation package written in C++ (Stroustrup 1985). The reducer also is operational on our 18-node BBN Butterfly multi-processor, using a "task pool" library task package (Tinker and Lindstrom 1987).

6.2 Normal Order Committed Choice In Practice

As we proclaimed from the outset, we do not consider Guarded SASL+LV *itself* to be a contribution to lan-

```
strict (defuse (merge [s1, s2]))

where
    s1 = [(0:1000), (1:2000), (0,3000)],
    s2 = [(0:2000), (0:3000), (1:4000),
          (1:1000), (1:3000)],

    defuse s = bind 1 s nil,

    bind n p Symtab =
       p = nil -> nil;
       build (f = 0 -> W; n)
       where
           build v =
              (f:v):(bind n+1 ptail
                          (enter (id:v) Symtab)),
           (f:id):ptail = p,
           enter (id:v) symtab =
                 lookup (id:v) symtab -> symtab;
                 (id:v):symtab,
           lookup (id:v) symtab =
                 symtab = nil -> false;
                 id1 = id -> fixdef v v1;
                 lookup (id:v) symtab1
           where
                 (id1:v1):symtab1 = symtab,
                 fixdef X X = true
```

Figure 8: Fair merge one-pass definition/use example.

guage design. Rather, it is a laboratory vehicle for investigating the compatibility and synergy of two very powerful semantic concepts: unification and normal order evaluation. Our studies are still underway, but the initial results have taught us that our initial enthusiasm should be tempered with some caution.

- *Advantages:* SASL+LV supports the composition of higher order functions on infinite data structures, while exploiting deferred binding and interprocess communication by monotonic refinement of shared data structures. This opens genuinely new possibilities for the design of declarative programs.
- *Disadvantages:* However, lazy evaluation can cause unenvisioned delays in the binding of logical variables. Indeed, since variable binding actions are a form of "genteel side-effects," they can make visible *when*, and in *what order*, subexpressions are evaluated. For example, in Fig. 8 the applicative order function `strict` (not shown) is necessary to prevent a deadlock in our implementation. Otherwise, the top-level (sequential) print function would await the binding of the first usage logical variable, without forcing its binding by evaluating the tail of the output string! Similarly, in the same figure, one might have alternatively defined `enter` as follows:

```
enter (id:v) symtab =
   symtab = nil -> nil;
   id1 = id -> fixdef v v1;
   (id1:v1):(enter (id:v) symtab1)
where
   (id1:v1):symtab1 = symtab,
   fixdef X X = symtab
```

However, this often fails to "set" the value of the final symbol defined, for the lazy top-level list construction in `(id1:v1):(enter (id:v) symtab1)` means the binding is never actually done unless the symbol is subsequently used!

7 COMPARISONS TO PREVIOUS WORK

7.1 Functional Languages With Logical Variables

Many languages combining functional and logic programming have been proposed, with varying degrees of semantic and implementational ambition (DeGroot and Lindstrom 1986). However, most do not directly correspond to Guarded SASL+LV, e.g. by admitting only deterministic parallelism, or by presuming only sequential evaluation. The closest examples are:

- Id Nouveau (Arvind et al. 1987, Nikhil et al. 1986, Nikhil 1987), which uses write-once *I-structures* as a form of logical variables in a dataflow framework;
- Kieburtz's F+L (Kieburtz 1986, Kieburtz 1987), employing equational clauses within function definitions to solve for logical variable bindings.

Danforth has critically examined this language design area as a whole (Danforth 1985).

7.2 Committed Choice Logic Programming

We have clearly benefited from the insights and experience of researchers working with the three best-known committed choice logic programming languages.

- *Concurrent Prolog:* Even in its "flat" form, Concurrent Prolog offers more power than Guarded SASL+LV through its ability to create shadow environments holding private copies of variables

conveyed into guards by goals. This permits certain categories of guards to succeed in CP that would fail, or deadlock, in Guarded SASL+LV. It is difficult to imagine such a mechanism in a distributed graph reduction environment such as we have described here. However, Concurrent Prolog has a genuinely more difficult problem in its requirement to export atomically all the bindings in a guard's shadow environment upon commitment (Taylor et al. 1986).

- *Parlog:* Guards in Parlog are required to be *safe*, i.e. not bind any variable in an input argument in the head of a clause. This is quite comparable to our trial unification policy. Indeed, Guarded SASL+LV might be viewed as an attempt to elevate Kernel Parlog to a normal order user-level language, without explicit mode annotations.

- *Guarded Horn Clauses:* Finally, we acknowledge our primary inspiration from Guarded Horn Clauses. Except for GHC's eagerness in initiating body trial evaluation concurrently with guard trial evaluation, Guarded SASL+LV conducts its commitment competitions in a manner very similar to GHC (Ueda 1986). Our implementation technique bears many resemblances to techniques developed for distributed implementations of GHC, e.g. (Ichiyoshi et al. 1987).

8 CONCLUSIONS AND FUTURE WORK

We have outlined the design, semantics, implementation, and user pragmatics of a normal order functional language enriched with logical variables and committed choice indeterminacy based on trial unification of goals. Our findings indicate that normal order and unification are a very powerful semantic combination, but raise delicate issues that must be investigated in a full language design and implementation before a conclusive appraisal can be made.

Our work will be continued in the following directions:

1. *Static analysis:* Fine grain `S`, `K`, `I` combinators are very attractive for prototyping systems, but have unattractive economic properties in larger scale usages. We have preliminary results on how static analysis can be applied to to the compilation of SASL+LV to larger grain combinators (Lindstrom 1986, Lindstrom et al. 1987, Lindstrom 1988). These techniques cluster program operators into larger granularity combinators through a static analysis technique that assesses both operator strictness (graph partitioning into co-evaluation equivalence classes) and mode effects on logical variables (e.g. "read-only" occurrences).

2. *Applications:* We plan more thorough appraisal of the pragmatic merits of Guarded SASL+LV through application to the construction of a system for concurrent object oriented programming (Kahn et al. 1986), and its demonstration for hardware description (Sheeran 1985). In the latter application, we view lazily evaluated streams of logical variables as potentially very attractive for modeling such phenomena as synchronous systems using pass transistors and arbitrated buses.

3. *Shadow bindings:* Finally, we aspire to incorporate some form of provisional binding environments local to guards, as per Concurrent Prolog.

ACKNOWLEDGEMENTS

We are indebted to Lal George for his tutelage on committed choice logic programming languages; to Peter Tinker for the Butterfly task library package; to John Evans for the *yacc*-based compiler front end, and to Dowming Yeh for his insights on static analysis.

This research was supported in part by grant CCR-8704778 from the National Science Foundation, and by an unrestricted gift to the University of Utah from Telefonaktiebolaget LM Ericsson, Stockholm, Sweden.

References

[1] Arvind, R. S. Nikhil, and K. K. Pingali. I-Structures: Data Structures for Parallel Computing. In Joseph H. Fasel and Robert M. Keller, editors, *Graph Reduction: Proceedings of a Workshop*, pages 336–369, Springer-Verlag, 1987. Lecture Notes in Computer Science No. 279.

[2] Göran Båge and Gary Lindstrom. *Combinator Evaluation of Functional Programs with Logical Variables.* Technical Report UUCS-87-027, Department of Computer Science, University of Utah, October 1987.

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

[4] D. DeGroot and G. Lindstrom. *Logic Programming: Functions, Relations and Equations.* Prentice-Hall, Englewood Cliffs, NJ, 1986. $39.95; reviews in *Computing Reviews* Aug. 1987, no. 8708-0643; SIGART Newsletter, July 1987, no. 101.

[5] 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.

[6] 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.

[7] Richard B. Kieburtz. Functions + Logic in Theory and Practice. February 25, 1987. 21 pp. unpublished paper.

[8] Richard B. Kieburtz. Semantics of a Functions + Logic Language. September 3, 1986. 17 pp. unpublished paper.

[9] 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.

[10] G. Lindstrom. Static Evaluation of Functional Programs. In *Proc. Symposium on Compiler Construction*, pages 196–206, ACM SIGPLAN, Palo Alto, CA, June 1986.

[11] 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.

[12] Gary Lindstrom. Static Analysis of Functional Programs With Logical Variables. In *Proc. International Workshop on Programming Language Implementation and Logic Programming (PLILP '88)*, pages L1 – L18, INRIA, Rocquencourt, France, March 16-18 1988. Edition Provisoire; to appear in Springer LNCS series.

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

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

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

[16] Keshav K. Pingali. *Demand-Driven Evaluation on Dataflow Machines.* PhD thesis, Mass. Inst. of Tech., Cambridge, Mass., May 1986.

[17] Keshav K. Pingali. Lazy Evaluation and the Logical Variable. In *Proc. Inst. on Declarative Programming*, Univ. of Texas, Austin, Texas, August 24-29, 1987.

[18] U.S. Reddy. On the Relationship Between Functional and Logic Languages. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations, and Equations*, Prentice Hall, 1986.

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

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

[21] 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.

[22] W. R. Stoye, T. J. W. Clarke, and A. C. Norman. Some Practical Methods for Rapid Combinator Reduction. In *Proc. Symp. on Lisp and Func. Pgmming.*, pages 159–166, ACM, 1984.

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

[24] 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.

[25] 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.

[26] D. A. Turner. SASL Language Manual. November 1983. Revision of August 1979 version.

[27] K. Ueda. *Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard.* Technical Report TR-208, ICOT, Tokyo, 1986.