IMPLEMENTING LOGICAL VARIABLES
ON A GRAPH REDUCTION ARCHITECTURE

Gary Lindstrom

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

Abstract

Logical variables offer a semantic meeting ground between functional and logic programming languages. That is, if functional languages are extended to do parameter passage by unification, much of the power of AND-parallel logic programming is obtained. However, multi-path search, by OR-parallelism or backtracking, remains the province of logic programming.

We outline an implementation strategy for FGL+LV, a functional language with logical variables, on the Rediflow multiprocessing graph reduction architecture. The aspects of logical variables receiving special consideration include:

a. parallel unification, especially proper treatment of indeterminate behavior, e.g. mutual exclusion on variable binding;

b. variable binding through emulated graph node merging;

c. exploitation of two levels of demand: assertive (during unification), and non-assertive (ordinary "read-only" usage), and

d. avoidance of meaningless cyclic variable bindings.

The existing base language implementation can be smoothly extended, with a word size increase of only one bit (needed to implement two levels of demand). Lazy evaluation in the base language is retained, except that actual parameters are now made strict to one level of evaluation. This requirement, overlooked in a previous paper on this subject, is argued to be semantically and operationally inescapable in a functional language with logical variables. It also provides insight into the vexing problem of how to apply the occur check in a language with infinite data objects. A novel technique for merging cyclic lists is used to implement logical variable binding in a distributed manner without locking or busy waiting.

\textsuperscript{1}This material is based upon work supported by NSF Grant DCR-8506000, and a Shared University Research Grant from the IBM Corporation.
1. Incorporating Logical Variables into Functional Programming

1.1. Essential Characteristics

From the perspective of functional programming, logical variables present a radically different basis for information flow. In particular,

- logical variables are not directly bound by uniquely determined expressions, in contrast to the customary dataflow orientation of functional languages;
- rather, they denote values determined by the cumulative application of successive constraints.

1.2. Advantages

In a functional programming language with logical variables, this difference can permit novel effects otherwise quite difficult to obtain. These include:

- an action at a distance effect, whereby certain kinds of "side-effects" can be achieved within an applicative setting (the Prolog "difference list" trick for constant time list concatenation is a familiar example);
- a stronger object orientation, under which logical variables play the role of mutable shared entities, and
- the use of this object orientation as a basis for isotropic process communication and synchronization, as in Concurrent Prolog [17, 18].

We investigate here the adoption of logical variables in a deterministic setting, i.e. without support for backtracking or OR-parallel search. This single extension brings into the realm of functional programming such elegant techniques as the Milner algorithm for polymorphic type checking [14]. And, while the absence of OR-parallelism may be lamentable, in partial compensation we retain functional programming’s execution directionality, which offers a clean solution to the AND-parallel control problem which plagues pure logic programming.

1.3. Challenges

If one seeks to add logical variables to a normal order functional programming language, the following challenges quickly become apparent:

a. semantic upward compatibility, including lazy evaluation;

b. preservation of concurrency opportunities;

c. assurance of overall determinacy despite aggressive exploitation of these concurrency opportunities, and

d. efficient implementation on distributed architectures.

A previous paper [13] addresses all but the last of these challenges, which is the subject of this paper.

1.4. Related Work

The task of amalgamating functional and logic programming is a highly active research area. Among the efforts most allied to the work reported here are:

- HASL [1], which adds conditional unification to a sequential form of SASL;