Abstract

Traditionally the design of programming languages has been compromised in order to exploit the most efficient machine architecture available, which has usually been of the von Neumann type. However, we believe that functional languages provide the most effective means for producing software and that the right approach is to develop a customised architecture for the implementation of the most suitable computational model for these languages.

In this paper we adopt graph reduction as the most natural computational model for functional languages and show how to represent it in a packet-based abstract computer architecture, ALICE. This architecture is highly parallel in nature, with a collection of processing agents performing redex reductions concurrently and asynchronously on the one hand, and the packets that represent the nodes in a function expression graph being distributed over several memory segments on the other. We suggest how a concrete parallel machine can be built to realise this abstract architecture, and describe the design of an experimental prototype machine. This prototype is a hardware emulator of the ideal concrete architecture which is to be built in customised VLSI, and is constructed using the INMOS Transputer as the basic building block. The design utilised a performance model to assist in choosing the optimal organisation of machine components, such as the ratio of processor to memory boards, and the model's predictions for the performance of various configurations of the machine are presented, along with some preliminary measurements made on the prototype. Finally, we make some suggestions for future design enhancements based on our experiences to date.

1. Introduction

We advocate the design of a machine tailored to executing programs written in functional languages. This contrasts with the historical approach where programming language design has been constrained by the von Neumann computer architecture. A natural computational model for the evaluation of functional expressions is graph reduction, in which the graph representing an expression is repeatedly reduced, by transforming sub-graphs which represent redexes (reducible expressions). At each stage there may be one or more redexes which can be reduced, the choice of the next redex or redexes being determined by the evaluation order (computation rule). Thus we may adopt for our customised architecture a parallel evaluator in which the tasks to be computed simultaneously are identified with the redex rewrites which occur in the graph reduction computational model. In this way the architecture can exploit the parallelism inherent in functional languages, and formal transformation offers the prospect of providing techniques to find equivalent functions for which parallelism can be exploited to a greater extent.

In the next section we explain the concept of graph reduction in the context of the functional language HOPE [BM80], and show how a packet-based machine architecture can be used to realise this computational model. This leads to an outline, at the abstract level, of the ALICE architecture, [DR81]. In section 3 we describe the design of the transputer-based prototype machine, before discussing its parameterisation, which employed an analytical model, in section 4. Performance issues and current status are addressed in section 5 before the paper concludes with an outline for future research and a summary in section 6.
2. A packet-based abstract architecture for graph reduction

2.1 Evaluation of first-order expressions

Graph reduction, corresponding to the evaluation of first-order functional expressions, proceeds by transforming the graph representing an expression with respect to some redex node which represents a function application (\(\beta\)-reduction in the case of \(\lambda\)-expressions). The node is then overwritten with the sub-graph which represents the result of the application. There are three cases to consider:

(i) If the associated function is primitive, and all of its required argument sub-graphs have been evaluated, i.e. cannot be further reduced, then the application is evaluated and the redex node is replaced by the root-node of its result. In the case of a primitive arithmetic function, for example, all the arguments must be evaluated, and the result-sub-graph consists of the single root-node which contains an item of atomic data. But in the case of the conditional function, only the predicate must be evaluated, and the result is the appropriate argument-sub-graph.

(ii) If the associated function is primitive, but not all of the arguments it requires have been evaluated, evaluation of the next required argument (specified in the delta rules of the primitive function) is initiated.

(iii) If the associated function is programmer-defined, the redex node is replaced by the sub-graph formed from that representing the function's body (i.e. its defining expression), but with each arc that is connected to a node representing a formal parameter redirected to the corresponding sub-graph representing the actual argument. In other words, the body is essentially copied with its formal parameters instantiated by argument expressions in the usual way.

In fact (iii) is a little more complex if the reduction machine performs pattern matching, as is the case with ALICE. Pattern matching requires that certain arguments have been evaluated at least far enough to permit comparison with the patterns. When they have not, analogues of the delta rules of primitive functions initiate such evaluation prior to reduction.

The selection of the next redex to reduce corresponds to the evaluation order (or computation rule), for example Normal Order Reduction (which is semantically safe) or Parallel Reduction. We will return to this point later in this section, but it should be noted that Parallel Reduction gives Normal Order Semantics (apart from the problems of halting unwanted non-terminating computations, if such are initiated).

We now give an example of a functional program, written in HOPE, which will be used to illustrate the operation of the ALICE machine and then show how an expression-graph can be represented as a collection of packets in the memory of an abstract architecture. Consider the data type "tree", and suppose we wish to write a function, "size", which takes a tree as argument and returns the number of data items held in the tree as its result. In HOPE this might be written as:

```hope
data tree(\(\alpha\)) == tip(\(\alpha\)) ++ node(tree(\(\alpha\)) x \(\alpha\) x tree(\(\alpha\)))
dec size : tree(\(\alpha\)) -> NUM

--- size(tip(i)) <= 1
--- size(node(t_1, i, t_2)) <= plus(1, plus(size(t_1), size(t_2)))
```