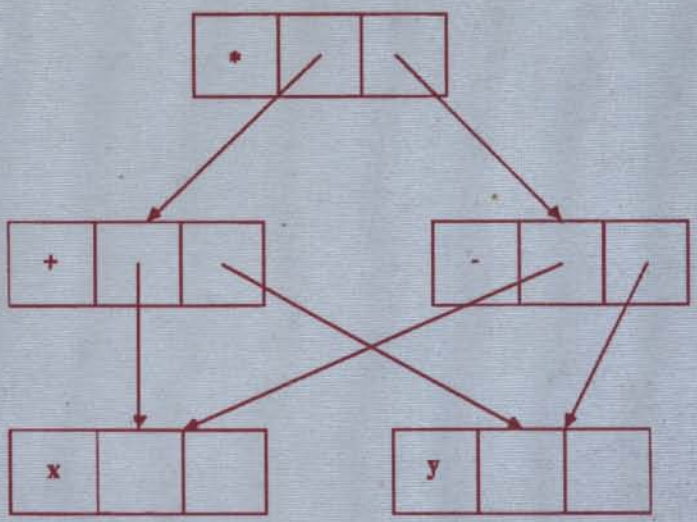
Lecture Notes in Computer Science 586

A. Cheese

Parallel Execution of Parlog



Springer-Verlag

Lecture Notes in Computer Science 586
Edited by G. Goos and J. Hartmanis

Advisory Board: W. Brauer D. Gries J. Stoer

A. Cheese

Parallel Execution of Parlog

Springer-Verlag
Berlin Heidelberg New York
London Paris Tokyo
Hong Kong Barcelona
Budapest

Author

Andrew Cheese
Siemens-Nixdorf Information Systems AG, Multiprocessor Unix Kernel Group
Otto Hahn Ring 6, W-8000 Munich 83, FRG

CR Subject Classification (1991): D.3.4

ISBN 3-540-55382-7 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-55382-7 Springer-Verlag New York Berlin Heidelberg

Typesetting: Camera ready by author
Printing and binding: Druckhaus Beltz, Hemsbach/Bergstr.
45/3140-543210 - Printed on acid-free paper

Preface

Logic programming has been proposed as a programming methodology that may help in producing more reliable and maintainable computing systems than those presently in service. At the same time there has been a trend for making faster computers by building machines using multiple CPUs. As a result of these two factors a family of concurrent logic languages has been developed to tackle the problems of programming these parallel computer architectures. It is important that the implementations of such computer languages should be efficient, otherwise the benefits of parallelism will be lost.

This monograph concentrates on the programming language Parlog and on computational models for its efficient execution. Two such models are developed, one a fine-grain Packet-Rewriting model and the other more coarse-grained, the Multi-Sequential model. Both models are reviewed in detail and software simulators have been built for them. Results from the simulations show that the Multi-Sequential model is very promising whereas the Packet-Rewriting model does not appear to suitable for the efficient execution of logic languages. These results have considerable importance for the design of parallel logic programming systems, and the implications are outlined and discussed in the concluding chapter.

I have many people to thank for helping me with this research. First and most of all, I would like to thank my supervisor, David Brailsford for all his help and encouragement during my research.

I am very grateful to the following people for proof-reading sections, and in some cases all, of this monograph and for their comments : Uri Baron, Steve Benford, David Brailsford, Mark O'Brien, Sergio Delgado-Rannauro, Dave Elliman, Colin Higgins, Graem Ringwood, Andy Walker, and Marion Windsor.

Marion Windsor, the secretary of the Department of Computer Science, University of Nottingham, was particularly helpful throughout the course of this research.

I would like to thank the following members of the Department of Computing Science at the University of Nottingham for providing a friendly and stimulating working environment, William Armitage, Peter Cowan, David Evans, David Ford, Eric Foxley, Godwin Gwei, Leon Harrison, Mike Heard, Roger Henry, Phillipa Hennessey, Emiko Hiraga, Kevin Hopkins, Anne Lomax, Graeme Lunt, Julian Onions, William Shu, Hugh Smith, and Mary Tolley.

Special thanks are due to Simone Mahlenbrei who helped me at various times whilst I was putting the finishing touches to this book at the European Computer-Industry Research Centre GmbH (ECRC).

This research was funded by the UK Science and Engineering Research Council (SERC).

February 1992 Andrew Cheese

Contents

Chapter 1 Introduction

Background

There is a growing recognition that there is a "software crisis" [3] in the sense that software systems are becoming too complex for the available programming languages and tools to handle. There are numerous computer-related disasters which lead one to believe that this is true [105]. These include reports of many Space Shuttle launch failures because of faulty software, the Vancouver Stock Index losing 574 points over 22 months as a result of a software rounding error, an *F18* aircraft crashing because of a missing exception condition, and *Viking* having a misaligned antenna caused by a faulty code patch.

The range of widely used imperative programming languages and the bad programming styles they tend to encourage are frequently cited as a major cause of faulty software. The meaning of the adjective "imperative" is "commanding". In fact, more often than not, the only means of understanding an imperative language program is as an ordered set of state-changing commands. This is because imperative programming languages are based on the von-Neumann model of computation, whereby a processing element is tightly-coupled with memory and executes a series of instructions guided by a program counter which indicates the "next instruction" to be obeyed. The notion of global memory is inherent in such programming language designs and together with the use of destructive assignment of new information to be stored can cause unforseen side-effects leading to obscure program bugs. An example of this might occur when a global and local variable differ by one character in their names and the global name is used by mistake for the local name. This error would remain undetected because the destructive-assignment statement is syntactically and semantically correct. In short, the solving of a problem with an imperative programming language requires not only a specification of *what* the solution is, but also a description of *how* to solve the problem.

The answer seems to lie in a paradigm which allows the programmer to state, declaratively, a description of a solution to the problem at hand and lets the target machinery perform the necessary computation. This philosophy has encouraged computer programmers to look towards the descriptive formalism of mathematical logic to help them achieve this goal. The result is a family of so-called declarative languages. In

fact there are two camps of declarative programmers who support either the logic programming or the functional programming style. There are fundamental differences in these two styles [44] but in both cases the resulting program consists of a set of assertive equations and computation is the deduction of some property with respect to these assertions.

Declarative language programs can be understood by static analysis because the meaning of any individual program segment is independent of the meaning of all the other textually separate parts. This, in turn, means that the semantics of the entire program is independent of the order of evaluation of these parts [97]. Therefore it is perfectly safe to evaluate declarative programs using a parallel computational model. All parallelism is implicit in the program, implying that the degree of concurrency exploitable is limited only by the degree of inherent parallelism present in the program.

Functional programming languages are directional; that is, the inputs and outputs of the relations they define are statically determined. As a result they utilise one-way pattern matching as a parameter passing mechanism. This is a drawback compared to the lack of modality (nonspecification of whether arguments are inputs or outputs) which is inherent in the logic programming paradigm. In the conventional imperative programming sense of parameter passing, this means that arguments can be used as either inputs or outputs, the correct mode being determined at runtime by the logic programming system and not by the user. An example of this property is the `append/3` (the notation `f/n` meaning `f` is a symbol of a structure of `n` arguments) relation allowing the user to specify a program which can be used both to concatenate two lists and to split a list into two sublists.

The functional paradigm does enable very powerful and flexible data type systems to be developed [16], allowing the user to define arbitrary types. The declarative and operational semantics of a functional programming language are based upon the lambda calculus [5] [25], and a reduction model of computation [172], respectively. A functional programmer will think in terms of functions and their evaluation. The most widely used of functional programming languages is the functional subset of Lisp [182]. However, there are now more powerful programming languages based on the lambda-calculus such as Miranda [167], SML [76] [55], and Haskell [171].

Logic programming languages are relational; a program is a conjunction of equations each expressing a relation between objects of interest to the programmer. Each equation is an implication stating that a property of an object or properties of a set of objects, are conditional on other object properties being true. In order to determine whether properties are true or false it is usual for the programmer to express some relations between objects that are vacuously true. Each equation is called a "clause" and a set of ordered clauses having the same relation name, "predicate symbol", is called a "relation" or "procedure".

Logic programming systems employ "unification" as a parameter-passing mechanism. As an example consider unifying the head of a clause `p(X, 2, Z)` with a goal `p(1, Y, 3)`, where a goal is analagous to a procedure call in conventional imperative programming languages with two-way parameter passing, and variables are denoted by lexical items beginning with a capital letter. The unification procedure is concerned with finding substitutions of variables to make a set of terms equal. In the example the substitution would be `{ X/1, Y/2, Z/3 }`, where the notation `V/t` denotes that the variable `V` is bound to the term `t`. Unlike the case of executing functional programming languages, it is possible to bind variables in the goal, e.g. `Y` in the above example. This enables logic programmers to make use of so-called "logical variables", that is, they are allowed to instantiate variables to terms which are non-ground i.e. the terms contain variables. For instance, consider the term `request(S)`. The variable `S` can be bound to another term `message(info, Answer)` which itself contains the variable `Answer`. This powerful feature enables a relation to only deal with the part of a data structure it is interested in, leaving "holes" to be filled in by other, more appropriate, relations. A practical use of this is in building one-pass compilers. Conventionally compilers written in imperative programming languages are two-pass. The first pass simply gathers up all the symbols in a program and makes a note of where they are in the program, so that on the second pass these symbol references can be filled in correctly. If the compiler is written in a logic programming language it suffices to leave the reference to the symbol as a variable which can be instantiated to the correct address later, when this has been determined. The procedural and declarative semantics of logic programming languages have foundations in theorem proving and predicate logic [99].

The declarative programming language paradigm has now been recognised as a viable alternative to conventional imperative approaches. As a result of this, several industrial organisations are now embarking on research programmes focusing on logic programming. Examples of these are the European Computer-Industry Research Centre (ECRC) in Munich, West Germany, funded jointly by Bull SA, Siemens and ICL [151] [54]; the Software Technology Division at MCC in Texas, U.S.A.; the Institute of New Generation Computer Technology (ICOT) which, although funded by the Japanese government, most of the researchers are from industry [63]; and the Swedish Institute of Computer Science (SICS) [176], which has been formed using 50% funding from the Swedish government but with help from industry. There is still a long way to go, however, before research results will reach the bulk of computer product consumers and until that time the majority of computer users may remain unconvinced of logic programming's potential.

Logic Programming

The state of logic programming is now arguably more advanced than that of the functional paradigm. This can be traced to the spread of Prolog, both in academic circles and industry [122] [123] [124] [125]. Logic programming language systems are more usable than their functional programming counterparts. Quintus Prolog for instance, provides a very good development environment incorporating modules, type-checking and other tools [121].

Theoretical Background

In order to describe the logic programming paradigm, it is necessary to review both the syntax and semantics of first-order predicate logic. Syntactically, first-order predicate logic is defined by a language over some alphabet. This alphabet consists of variables, constants, functions, predicates, connectives, quantifiers and punctuation symbols. The last three classes of symbols are the same for any formula of the logic. The connectives are $\neg$, $\wedge$, $\vee$, $\rightarrow$ and $\leftrightarrow$. Their intended meanings are "negation", "conjunction", "disjunction", "implication" and "equivalence". The quantifiers are represented by the symbols $\forall$, meaning universal quantification i.e. "for all", and $\exists$ meaning existential quantification i.e. "there exists". The punctuation symbols are "(", ")" and ",". Variables are normally represented by capital letters, e.g. X, Y, and Z. Constants are

represented by letters near the start of the Roman alphabet such as a, b and c. Function symbols are denoted by letters chosen from a third of the way through the Roman alphabet such as f, g, and h. Predicate symbols are denoted by letters chosen from two-thirds of the way through the Roman alphabet, e.g. p, q, and r.

The terms of first-order predicate logic are defined inductively. All variables are terms, all constants are terms, and if f is a function (or functor) of arity n and t1, ..., tn are terms then f(t1, ..., tn) is a term. The well-formed formulas of first-order predicate logic are also defined inductively. If p is a predicate symbol of arity n and t1, ..., tn are terms, then p(t1, ..., tn) is a formula. In this specific case the formula is called an "atom" (in the logic programming world, as opposed to theoretical logic studies, atom also refers to a constant). If F and G are formulas then so are ~F, F $\wedge$ G, F $\vee$ G, F $\rightarrow$ G, and F $\leftrightarrow$ G. If F is a formula and X is a variable then $\forall$X F and $\exists$X F are formulas.

In the quantified formulas $\forall$X F and $\exists$X F the variable X is said to be "bound". In general, all occurrences of a variable following a quantifier which are also present in the quantified formula are "bound". Any variable present in a formula which is not bound is said to be "free". For instance, in the formula $\forall$X p(X, Y), X is bound and Y is free. A formula with no free occurrences of variables is said to be "closed".

A "literal" is an atom or the negation of an atom. A "positive literal" is simply an atom and a "negative literal" is the negation of an atom. A "clause" is a closed universally quantified disjunction of literals. That is, it is of the form

$\forall$X1 ... $\forall$Xn (L1 $\vee$ ... $\vee$ Lm)

where each Li is a literal and each Xi is a variable. Logic programs consist of a set of clauses. Thus a special notation can be adopted to make programs easier to read. The clause

$\forall$X1 ... $\forall$Xn (A1 $\vee$ ... $\vee$ Aj $\vee$ ~B1 $\vee$ ... $\vee$ ~Bk)

can be represented in this new semantically equivalent clausal form notation as

A1, ..., Aj $\leftarrow$ B1, ..., Bk

where all the variables, X1, ..., Xn, are assumed to be universally quantified. The

commas in the consequent, A1, ..., Aj, denote disjunction and the commas in the consequent, B1, ..., Bk, denote conjunction.

A "program clause" is a clause containing exactly one positive literal and takes the form A ← B1, ..., Bk. A is called the "head" and B1, ..., Bk the "body" of the program clause. A "unit clause" is a program clause with an empty body, that is, it takes the form A ←. The informal meaning of a program clause such as A ← B1, ..., Bk is that if for every assignment of each variable in the above clause, the conjuncts B1, ..., Bk are true then A is true. Program clauses with non-empty bodies are "conditional" and unit clauses, with empty bodies, are "unconditional". The informal intended meaning of the unit clause A ← is that, for each assignment of each variable in the clause, the literal A is true. A "goal clause" contains no positive literals and is of the form ← B1, ..., Bk. Each Bi is called a "subgoal" of the clause. The "empty clause", with no antecedent or consequent, is understood as meaning a contradiction. A "Horn clause" is either a program clause or a goal clause, that is, there is at most one positive literal present.

Most logic programming language dialects are based upon the Horn clause subset of first-order predicate logic. One of the properties of first-order predicate logic is that it has equivalent operational and declarative semantics. The procedural meaning of a theory is given by a "proof theory", that is, it is a corresponding "proof tree". A proof may proceed by negating the formula that is to be proved and trying to obtain a contradiction using the clauses making up the program, thus concluding that the original unnegated formula is a logical consequence of the program.

A proof may use use of a procedure known as "resolution" to construct its derivation. A logic programming system makes use of resolution to try and reduce the given query (a goal clause, in fact the negation of what is actually to be proved) to the empty clause. It does this by first selecting a literal from the goal clause and attempting to unify it with the head of a program clause. If this succeeds the original literal is replaced by all the body literals giving a new goal clause. The unifying substitution is then applied to this new clause and the whole process reiterates. If the attempted unification should fail, the system will attempt to select a literal again. This chosen literal could be the same one again, in which case an alternative clause for the predicate would be used, and unification attempted. This whole process repeats until the empty clause is derived or until it is

discovered that there is no route available for the computation to proceed and it would then be concluded that the original goal clause is not a logical consequence of the program. The current goal clause is called the "resolvant".

Consider the effect of resolution on the goal clause and program clauses given below.

```
← p(X, 1), q(X, Y).

p(2, Z).
q(1, 1).
q(2, 2).
```

The first thing the resolution process does is to select an atom from the current goal clause. In the programming language Prolog the leftmost atom is chosen. Assume then that this is the "selection function" used, meaning that the atom `p(X, 1)` is selected. This unifies with the first program clause giving a substitution `{X/2, Z/1}` which means that the variables `X` and `Z` are bound to the constants `2` and `1`. Applying this substitution to the current goal gives us a new resolvant `q(2, Y)`. There are two program clauses defining the procedure `q/2`. Unification of the current goal with the first of them fails because the constants `1` and `2` are unequal. Unification does, however, succeed using the second clause for `q/2` giving a substitution `{Y/2}`. The current goal is now empty and the resolution procedure succeeds and terminates.

An "interpretation" of a logic program consists of some domain of discourse over which variables can range. All constants of the program are assigned an element of the domain. All functions are assigned a mapping over the domain. All predicates are assigned a relation on the domain. All quantifiers and connectives have an *apriori* fixed meaning. Thus an interpretation is used to give meaning for every symbol in a program. An interpretation in which a formula expresses a true statement is said to be a "model" of the formula. The programmer usually has an interpretation in mind when writing a program. This is the "intended interpretation", which, if it meets the specification, will be a model.

Consider the formula $\forall X \exists Y$ `p(X, Y)` and the interpretation *I* where the domain of discourse is the non-negative integers and `p` is assigned the relation $<$. *I* is then a model for the formula, as the formula expresses the true statement that "for every non-

negative integer, there exists another non-negative integer which is strictly greater than the chosen integer".

The declarative semantics of a logic program is given by characterising a particular model of the program. Intuitively, this is the one containing the minimal amount of information whilst still remaining a model. This is because any extra information in the model can only reflect on formulas which are not derivable from the program and thus, are irrelevant. This model is known as the "least model".

A set of clauses S is said to be "satisfiable" if there is an interpretation which is a model for S. Assume a logic program P and a goal clause G, then the problem is determining the unsatisfiability of $P \cup \{G\}$. This would seem to imply that every interpretation of $P \cup \{G\}$ must not be a model. There are, however, an infinite number of possible interpretations and it would not be feasible to verify that all of them give rise to unsatisfiability.

Fortunately, it is possible to identify a smaller class of interpretations which need only be considered. These are known as "Herbrand interpretations" [77]. Informally a Herbrand interpretation is one in which the domain of discourse consists of all the symbols present in the program. Each symbol maps to itself, and variables range over the symbols, and terms which can be constructed from these symbols.

Consider the program below

```
p(X) ← q(f(a),g(X)).
r(Y) ←.
```

We now give its Herbrand Interpretation. The domain of discourse is the set

```
{a, f(a), g(a), f(f(a)), f(g(a)), g(f(a)), g(g(a)), ...}
```

Constants are assigned themselves; thus a is assigned a. The functions f and g are assigned themselves taking arguments from the domain of discourse.

This property effectively means that deduction systems only have to work at the level of symbol manipulation and do not have to worry about what the user might have intended the symbols constituting the program to mean.

A Herbrand Interpretation for a program which is a model is a "Herbrand Model". The meaning of a logic program should consist of just those formulas which are logically

implied by the program. The declarative semantics of a program are therefore given by the "least Herbrand model". This consists of just those atoms which are logical consequences of the program.

This underlying mathematics provides a formal basis for the construction of correct programs. This in turn implies there are sound methods for proving properties of programs and enables transformation methods to be developed. A typical application is the transformation of a program, written merely as a prototype specification of a solution to a problem, into a more efficient semantically equivalent one.

The standard unification procedures utilised by most Prolog implementations are purely syntactic. Terms are equal if they are syntactically identical with the appropriate unifying substitution applied. However, because of logic programming's mathematical foundations, it is possible to develop new types of languages called "constraint logic programming languages" [173] [82] [50] [51], which enable the user to supply a constraint-solver which works in the user's intended domain of interpretation, for example real numbers. This, together with the current work on implementation, will lead to the development of more powerful logic programming systems than those around today.

Developments in Prolog Implementation

The development of logic programming owes much to the programming language Prolog [34]. In 1972 Phillipe Roussel designed the first Prolog interpreter at the Université d'Aix Marseilles. In fact, the name Prolog was suggested by Roussel's wife Jacqueline, as an abbreviation for *programmation en logique*. It was written in Algol-W and employed the clause-copying technique. This means that whenever a clause is selected as a candidate for reduction, a copy of it is made. This is a simple, albeit inefficient, method of avoiding name clashes with variables. Roussel then visited Edinburgh and learned of Boyer and Moore's structure-sharing approach for representing data structures [11]. Using this method, data structures are represented as skeletons with an associated set of variables which can be instantiated later. This way the skeleton of a data structure can be shared leaving the creation and binding of variables as the only expense. On returning to Marseilles, Roussel started work together with two of his students, H. Meloni and G. Battani, on a Fortran version of the original interpreter using structure sharing [132].