Anne Mulkers

Live Data Structures in Logic Programs

Derivation by Means of Abstract Interpretation

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

first layer, consisting of the *type* and *mode* analysis, basically supplies the logical terms to which variables can be bound. The two subsequent layers of the analysis heavily rely on these descriptions of term values. The *sharing analysis* derives how the representation of logical terms as structures in memory can be shared, and the *liveness analysis* uses the sharing information to determine when a term structure in memory can be live.

Acknowledgments

This book is based on my Ph.D. dissertation [59] conducted at the Department of Computer Science of the K.U.Leuven, Belgium. The research presented has been carried out as part of the RFO/AI/02 project of the *Diensten voor de programmatie van het wetenschapsbeleid*, which started in November 1987 and was aimed at the study of implementation aspects of logic programming: 'Logic as a basis for artificial intelligence: control and efficiency of deductive inferencing and parallelism'.

I am indebted to Professor Maurice Bruynooghe, my supervisor, for giving me the opportunity to work on the project and introducing me to the domain of abstract interpretation, for sharing his experience in logic programming, his invaluable insights and guidance. I wish to thank Will Winsborough for many helpful discussions, for his advice on the design of the abstract domain and safety proofs and his generous support; Gerda Janssens for her encouragement and support, and for allowing the use of the prototype for type analysis as the starting point for implementing the liveness analysis; Professors Yves Willems and Bart Demoen, for managing the RFO/AI/02 project and providing me with optimal working facilities; Professor Marc Gobin, my second supervisor, and Professors Baudouin Le Charlier and Danny De Schreye, for their interest and helpful comments, and for serving on my Ph.D. thesis committee. I also want to thank my family, friends and colleagues for their support and companionship.

Leuven, March 1993 Anne Mulkers

Contents

Chapter 1

Introduction

In conventional languages, such as C or Pascal, the programmer explicitly controls the utilization of memory by means of declarations and destructive assignments. For example, when reversing a linear list L, the list cells of the original list can be reused to construct the reversed list in the case that the original list is no longer needed for further computations. It is up to the programmer to decide whether he needs to preserve the old list intact and construct a reversed list which has only the list *elements* in common with the list L (e.g. Rev_L1 in Figure 1.1), rather than reuse the list-constructor cells of L as well (e.g. Rev_L2).

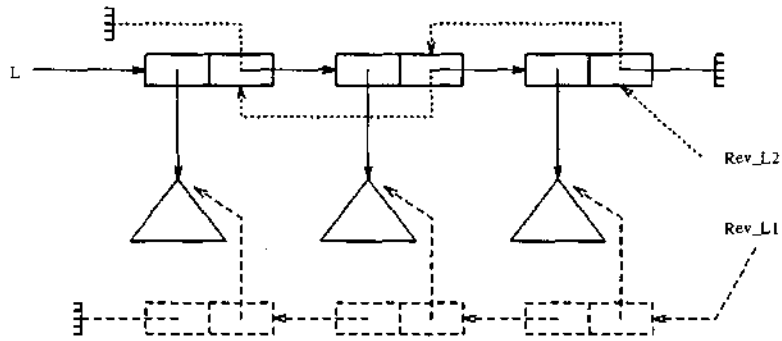

Figure 1.1: Reversing a linear list.

Applicative languages, in their pure form, do not have destructive assignments. Also type declarations are often absent. The declarative nature of these languages is often cited as an important advantage, which allows programmers to focus on the logic of the problems they have to solve, rather than on more technical aspects such as search control and efficient memory usage. Unfortunately, the performance of current implementations of applicative languages does not compare well with procedural languages yet. To achieve better utilization of memory, global flow analysis techniques are being developed that are concerned with determining the type and liveness of data structures that are dynamically

```
append(nil,_Y,_Y).
append([_E | _U],_Y,[_E | _W]) :- append(_U,_Y,_W).

nrev(nil, nil).
nrev([_E | _U], _Y) :- nrev(_U, _RU), append(_RU, [_E], _Y).
```

Program 1.1: nrev/2 (Naive reverse)

created during program execution. Knowledge about the lifetime of data structures guides the compiler in the generation of target code to reuse heap storage that is no longer accessible from program variables, i.e. to introduce destructive operations and avoid the copying of data structures that have no subsequent references.

In this book, we address the problem of liveness analysis for the class of pure Horn clause logic programs. The language considered has a countable set of variables (Vars), and countable sets of function and predicate symbols. A *term* is a variable, a constant, or a compound term $f(t_1, \ldots, t_n)$ where f is a n-ary function symbol and the t_i are terms. An *atom* has the form $p(t_1, \ldots, t_m)$ where p is a m-ary predicate symbol and the t_i are terms. A *body* is a (possibly empty) finite conjunction of atoms, written $A_1, \ldots, A_n$. A *clause* consists of an atom (its head) and a body and is written `A :- B`. A *program* consists of a finite number of clauses. A *query* or *goal* consists of a body only, written `?- B`. We assume that the reader is acquainted with the basic terminology of logic programming and the execution mechanism of Prolog which is based on unification and backtracking. Features such as *asserl* and *retract* are not considered, i.e. we assume that any source code for the predicates that can be executed at run time is available to the compiler.

The handling of data structures is very flexible in Prolog. Data manipulation (record allocation as well as record access and parameter passing) is achieved entirely via unification. An optimizing compiler can translate general unification to more conventional memory manipulation operations if information is available about the mode of use of the predicates. When at run time a compound term becomes accessible for the first time, we can say the term is being constructed. When a pattern is matched against a compound term that is already accessible, we can say the components of the term are being selected. Integrated type and mode analysis in many cases allows to predict at compile time whether a unification is a selection rather than a construction operation. Selection statements in particular are good candidates to check for the possible creation of *garbage cells*, i.e. cells that have no further references.

Consider the Prolog Program 1.1 for naive list reversal. We use the convention that variable names start with an underscore. If we assume that queries to nrev/2 are restricted to have as first argument a list that is no longer referenced after the call, and as second argument a free variable to return the output, then it is possible to generate target code for this program that allocates no new

list-constructor cells, but rather reuses the list cells of the first argument. Indeed, under the assumption, the integrated type and mode analysis will infer that each call to the recursive clause of nrev/2 has as its first argument a list, and as second argument a free variable. The unification of the call with the clause head *selects* the head and tail of the first argument list. The principal list-constructor cell of this list on the contrary has no subsequent references in the clause following the unification of the call with the clause head. This means that the compiler can recognize the principal list cell as garbage and generate target code that reuses it. For instance, consider the call to append/3 made by the same clause. A single element list [_E] needs to be constructed. Instead of allocating a new cell, the compiler can reuse the garbage cell that was detected.

Note that the problem is more complex if there may be multiple references to the cells of the input list. Most implementations of unification unify a variable and a compound structure by making the variable a reference to the structure – not a copy of the structure. The representations in memory of the logical terms to which variables can be bound typically share some of their structure: while the denoted terms make up a forest of trees, their representations form a more general directed acyclic graph. This is why in general the sharing analysis plays a crucial part in the liveness analysis.

In the above example, we can also infer that, the first two arguments in each call to append/3 will be lists and that the third argument will be a free variable. Again, it is possible to detect that, after invocation of the recursive clause of append/3, the principal cell of the first argument is garbage and can be reused to construct the value of the third (output) argument. Thus, all list constructions in this example can reuse garbage list cells, eliminating all allocation operations. Since the reused cells would otherwise be garbage, we have eliminated the garbage-collection overhead associated with the nrev/2 procedure. Moreover, a compiler can detect that the element field of each reused list cell already contains the value desired in the cells new use. The operations filling in these car fields can be eliminated from the generated target code. The resulting code closely resembles how a programmer using an imperative language would solve the problem of reversing a linear list of linked records.

In the present work, we propose an abstract domain and operations to analyze the liveness of data structures within a framework of abstract interpretation. Chapter 2 presents the principles of abstract interpretation for logic programs, and the application of type and mode analysis on which the domain for liveness analysis is based. In Chapter 3, we discuss work related to the application of compile-time garbage collection in the context of both logic and functional programming languages. In Chapter 4, we formalize an abstract interpretation for analyzing how the terms to which program variables are bound at run time, can share substructure in storage. We also augment the usual concrete semantics with information about sharing of term structures and discuss whether any implementation commitments are implied. As argued above, the sharing analysis constitutes a prerequisite for the liveness analysis. The latter is presented in Chapter 5. In both Chapter 4 and 5, the emphasis is mainly on the precision

and on the soundness of the results that can be obtained, rather than on the efficiency of the analysis. Due to imprecision that is inherent to the global analysis algorithms, not all garbage cells can be detected in arbitrary cases. We will extensively discuss the strength of the analyses that are proposed.

The study of code optimization schemes that explicitly reclaim or reuse garbage cells is beyond the scope of the present book. In [52], Mariën et al. discussed some preliminary experiments on code optimization based on liveness information. Only opportunities for *local* reuse of storage cells are considered, i.e. reuse within the same clause where a cell is turned into garbage. Non-local reuse would require extra run-time data areas to keep track of the free space. Although possible in principle, non-local reuse therefore will be less beneficial for code optimization. The reuse of storage also introduces some new requirements on the trailing mechanism of standard Prolog implementations that will affect the performance. We will briefly discuss these issues in Section 3.3 and 5.3.3.