Static Analysis of Logic Programs to Detect Run-Time Garbage Cells

A. Mulkers
Dept. of Computer Science
K.U. Leuven
B-3001 Heverlee, Belgium

W. Winsborough
Dept. of Computer Science
PennState University
University Park, PA, 16802

M. Bruynooghe
Dept. of Computer Science
K.U. Leuven
B-3001 Heverlee, Belgium

Abstract

A common hazard for implementations of applicative programming languages is the excessive creation of garbage cells during program execution. Run-time garbage collecting processes constitute a costly remedy for this fault. We developed a detailed analysis for logic programs that allows compiler optimizations to safely reuse space allocated to data structures. The analysis is based on abstract interpretation and its main contribution consists of extending an abstract domain and operations for type and sharing analysis such that when supplied to an abstract interpretation framework, the domain induces a powerful liveness analysis. Preliminary results obtained by an experimental prototype, confirm that useful and detailed information about the lifetime of term substructures can be derived.

1 Introduction

Typical implementations of applicative programming languages keep the stored representation of a data structure intact when its parts are used to construct a new data structure. Unless this is necessary because the old structure has a subsequent reference, a substantial overhead is incurred, as it is usually cheaper in time and memory usage to reuse the old value's storage. Addressing logic programs, this paper surveys a technique based on abstract interpretation to gather the (liveness) information needed by a compiler to reuse such storage in a safe way, i.e., to perform optimizations such as compile-time garbage collection [3], local reuse within a procedure [7], and general copy avoidance [8].

A wide variety of abstract interpretation frameworks have been constructed for the specification and verification of analyses of logic programs [2, 6, 13, 17, 18, 22]. Essentially, a framework is a parameterized

construction of an analysis together with theorems that ensure the soundness (conservative over approximation of the reachable states) and termination of the analysis constructed provided the domains and operations supplied to complete the construction obey certain safety requirements. When *concrete* domains and operations are provided, the induced analysis precisely characterizes the execution behavior of the program, but may not terminate. The task of constructing an analysis by using the framework is accomplished by supplying *abstract* domains and operations that obey the safety requirements, capture the information of interest, and, by properties of the domain, induce a terminating analysis. So called *top-down* analyses characterize the states in which clauses and their calls can be reached and require that some form of *query specification* be provided to characterize the entry uses of the program.

The present paper builds on prior work in liveness analysis for logic programs. [3] discusses the kind of analysis needed to enable compile-time garbage collection. Providing a key prerequisite of liveness analysis, [19] formalizes an *abstract interpretation* (a domain and unification operation) for analyzing how the terms to which program variables are bound at run time can share substructure in storage — so called, *sharing analysis*.

The research presently reported consists of extending the abstract interpretation for sharing analysis to an abstract interpretation for liveness analysis and of evaluating this new analysis in light of our experience with a prototype implementation. (Some experiments concerning the potential utility of liveness analysis for an optimizing compiler of logic programs have been reported in [16].) Space considerations do not allow us to include formal constructions or proofs [20]. Instead we illustrate the techniques we have developed and assess how they have performed in preliminary trials.

The paper is organized as follows. In Section 2 we motivate our work by illustrating how liveness-analysis results can be used by a compiler. Using a simple example, we show how cells identified as garbage can be locally reused in subsequent construction operations. In Section 3 we illustrate how liveness analysis can be performed and how it relies on sharing analysis. We report our experience with the prototype implementation in Section 4. Section 5 contains some concluding remarks.

2 Local reuse in Prolog

For the purpose of illustration, consider the Prolog program for naive list reversal, shown in Figure 1. If

```
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).
```

Figure 1: Naive Reverse

we assume that queries to `nrev/2` are restricted to have the form `nrev(Input,Output)` where `Input` is a list of known length that is no longer needed after the call (e.g., a list of integer elements) and `Output` is a free variable for returning the output, then it is possible to generate code for this program that allocates no new list-constructor (`cons`) cells but rather reuses the list cells of `Input`. To see this, first consider the recursive clause for `nrev/2`. Under the assumption, each call to this clause will have as its first argument a list. Moreover, the principal cell of this list has no subsequent reference following the unification of the call with the clause head. The compile-time detection of such facts is the subject of this paper. Given this information, the compiler can recognize the principal list cell as garbage and generate code that reuse it. For example, 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.

Under the assumption stated above, we can also infer that the first two arguments in each call to `append/3` will be input lists and that the third argument will be a free variable. Moreover, 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 associated allocation operations. Since the reused cells would otherwise be garbage, we have also eliminated all garbage-collection overhead associated with the `nrev/2` procedure. Finally, a compiler can detect that the element (`car`) field of each reused list (`cons`) cell already contains the value desired in the cells new use. Thus, the operations filling in these `car` fields can also be eliminated from the generated code.

Although the above example does not illustrate the point, it bears mentioning that this reuse of storage introduces some new requirements on the trailing mechanism of standard Prolog implementations. The trail records actions that must be undone explicitly on backtracking. Our notion of liveness considers only forward execution of the subject program, completely ignoring backtracking. The idea is that, when reusing a garbage cell, we replace the traditional WAM instruction “putstructure”, which allocates a cell on top of the heap, by “useoldstructure”, which takes the address of the cell to be reused as an argument. Before overwriting the cell, the new instruction performs a trail test and, if necessary, trails not only the address but also the content of the field to be overwritten, so that the cell can be completely restored on backtracking. Such an extension of the trail has been proposed, for example, to assist implementation of delay mechanisms [1, 21].

3 Liveness propagates among shared structures

In Section 2 we discussed the reuse of cells after they have been inspected based on the knowledge that there will be no subsequent reference to the cell. Of course, structure cells are not necessarily garbage immediately after their first inspection. Suppose the current resolvent has the form

$$\leftarrow (A_i, A_{i+1}, \dots, A_q)\sigma$$

where $A_i, \dots, A_q$ are the pending calls — either query calls or renamed copies of calls appearing in the bodies of the program clauses — and the substitution σ is the current variable-binding environment. Suppose the first argument of A_i is X and σ binds X to a structure, such as a list or a tree. Clearly, if X also occurs in $A_{i+1}, \dots, A_q$, then the structure to which X is bound is live and its storage must not be reused. However, this is not the only circumstance in which the structure is live. Most implementations of unification unify a variable and a structure by making the variable a reference

Figure 2: The proof tree of a derivation from the query “ $?-p([2,3], \text{Answer})$.”

to the structure — not a copy of the structure. Thus, 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. In general, a cell in the structure to which X is bound is live if any variable occurring in $A_{i+1}, \dots, A_q$ is bound to a structure that contains the cell in question. Viewed in terms of the represented logical terms, liveness propagates through the sharing of structure in the terms’ representations. Consequently, shared-structure analysis is a prerequisite of liveness analysis.

We illustrate this point by considering the clause

$p(L, \text{Out}) :- \text{append}([1], L, R),$
 $\text{append}(R, [4], \text{Out}), \text{use}(L).$

The proof tree of a derivation from the query “ $?-p([2,3], \text{Answer})$.” is shown in Figure 2. In the figure, clause variables are renamed apart (to avoid variable capture) in each clause invocation by adding a subscript corresponding to that invocation. The labels (1), ..., (4) indicate points in the computation where we consider snapshots of the local clause environment. The snapshots shown in Figure 3 represent the precise (concrete) environments. We picture terms as labeled trees to facilitate illustration of the instrumentation that captures sharing and liveness. The form of the environment as a tuple of variable bindings is a convenient formalism used extensively in [19], which we adopt here to ease cross references by the reader. In the figures, program variable names label the roots’ outarcs. Shared substructures are shown as connected by dotted arcs. Corresponding substructures of shared structures also share; however, we show

only the principal sharing arcs to avoid cluttering the figures. Variables are represented by nodes labeled V , with aliases connected by sharing arcs. Nodes decorated with $*$ are locally live because of a subsequent reference in the current clause instance, or sharing with such a node. Nodes decorated with $**$ are globally live because of a subsequent reference in an ancestor clause instance, or sharing with such a node.

For example, the structures of R_1 and L_1 are locally live at point (1) because they occur in the second and third call, respectively, of that clause instance while Out_1 , being the query’s intended solution, is globally live. The fact that Y_2 is globally live at point (2) is inherited from the liveness of L_1 at point (1). This illustrates how local information about the future need to access a structure is inductively passed down the proof tree to the liveness environments at lower levels of the proof tree, where the information can be used to determine which storage becomes garbage after it is referenced. At point (4), the only remaining reference to R_1 is in the next call, so R_1 is no longer locally live; however, the tail of R_1 is live because it shares with the structure of L_1 .

Our liveness analysis is constructed in three layers. The first layer monitors simply the logical terms to which variables can be bound; the second, how the representation of those terms as structures in memory can be shared; the third, when a structure in memory can be live. Each layer depends only on those that precede it. Our discussions behave as though all three layers are computed simultaneously, though they could be computed in turn.

In [19] we formalized a domain and unification operation for constructing the shared-structure information at both the concrete and the abstract levels. A *concrete sharing environment* has the form $\langle \mathcal{K}, \text{CSharing}_{\mathcal{K}} \rangle$ where $\mathcal{K}$ is the tuple of variable bindings and $\text{CSharing}_{\mathcal{K}}$ is essentially an equivalence relation over the substructures in $\mathcal{K}$, indicating which subterm occurrences are represented by the same structure in storage (shown by the dotted arcs in Figure 3). An *abstract sharing environment* represents a set of concrete sharing environments and has the form $\langle \mathcal{T}, \text{ASharing}_{\mathcal{T}} \rangle$. Here $\mathcal{T}$ is a type graph [11] representing a set of variable-binding tuples and $\text{ASharing}_{\mathcal{T}}$ is a set of pairs of nodes in the type graph representing possible sharing of structure.

[19] describes unification between two variables in an environment at the level of concrete sharing environments and at the level of abstract sharing environments and shows that the outcome of the latter

Figure 3: The concrete liveness environments

Figure 4: The abstract liveness environments

is a safe approximation of the unifications in the represented set of concrete sharing environments. The concrete unification captures all sharing that arises in an implementation that introduces sharing only when unifying a variable and a structure and that shares the representation of a variable among each of its occurrences. Standard implementations based on either the structure-sharing or structure-copying representation of terms meet this description. This includes WAM-based implementations.

We illustrate the abstract sharing environments, as well as the additional liveness information that we presently introduce, by showing in Figure 4 the abstract snapshots for the labeled points in the derivation shown in Figure 2. The type graph for L_1 at point ① represents the set of non-empty integer lists. As in the concrete case, the tail of R_1 is live at point ④ and cannot be reused in the subsequent call to `append`, while the storage occupied by the second argument of `append` would be a candidate for reuse, should the opportunity arise. (In the example, the opportunity does not arise.)

The current research extends the concrete and abstract sharing environments to include the liveness information, forming what we call *liveness environments*. The new concrete and abstract environment components are called, respectively, $CLive_K$ and $ALive_T$, and are given by sets of, respectively, subterm occurrences and type graph nodes. In the example, these are the nodes decorated with $\star\star$ and their descendants.

4 Experience

Recently, we have extended a prototype for type analysis [11] to perform sharing analysis and liveness analysis. We have used it to analyze a set of small and well-known programs manipulating data structures. The query specifications for these programs state the type of the input arguments, and the absence of sharing between different input components; they also state that the output arguments are free variables that are live (need to be preserved as output). Several list processing programs have been analyzed, e.g. `quicksort/2` (sorting a list – entails also analyzing `partition/4`), `mergesort/2` (sorting a list – entails also analyzing `split/3` and `merge/3`).

The *depth restriction* controlling the folding of the type graphs, used for *normal-form* programs wherein parameter passing is separated from unification, was two (i.e. at most two nodes on a non-circular path in

the graph can have the same label). The precision of the results was sufficient for the prototype analyzer to recognize that the data structures being consumed are turned into garbage. As these programs offer the opportunity to reuse these garbage cells, this information means that the programs can work in-place; the only heap-memory required is that occupied by the input data structures. We have also applied our prototype to analyzing tree-manipulating programs, such as `buildtree/2`, which transforms a list into a sorted binary tree. The analysis detects when list cells become garbage, however these cells cannot be reused, as tree cells require more storage space. Analyzing `buildtree/2` entails also analyzing `insert/3`, which inserts one element into a binary tree. Our prototype's analysis of `insert/3` indicates that the tree can be modified in place. Interesting enough, this results in the same memory usage as the skillful programmer obtains by using open-ended trees. (To achieve the same speed, the compiler must recognize when reused cells already contain the correct values and need not be initialized.)

5 Conclusion

Substantial prior work has been reported regarding liveness analysis for functional programming languages [4, 9, 10, 12, 15] but it does not transfer effectively to liveness analysis for logic programming. The problem is significantly different in the context of logic programming, primarily because the sharing introduced by unification is significantly more complicated to analyze at compile time than parameter passing, term construction and subterm selection.

A different approach to liveness analysis for the significantly restricted case of so-called *ground Prolog* has been developed in [14]. That work also does not handle general unification.

In previous work [19], we defined an abstract domain and unification operation for sharing analysis of logic programs. In the present paper, we sketch a method whereby the abstract domain is extended to perform liveness analysis in addition to sharing analysis and the primitive operations to compute the liveness information at both the concrete and the abstract levels are illustrated. Our analysis is constructed in layers. By this approach we have overcome the difficulty of analyzing liveness in the presence of full unification, where the sharing introduced depends on the structures being unified. Finally, we discussed the effectiveness of a prototype implementation on a small collection of well-known example programs.

Acknowledgements

A. Mulkers is supported by DPWB contract RFO/AI/02. W. Winsborough is supported in part by the National Science Foundation (USA) grant CCR 88-05503. M. Bruynooghe is supported by the Belgian National Fund for Scientific Research.

References

- [1] Boizumault, P., PROLOG l'implantation, Masson, Paris, 1988.
- [2] Bruynooghe, M., A Practical Framework for the Abstract Interpretation of Logic Programs, *Journal of Logic Programming*, 10(2): 91-124 (1991).
- [3] Bruynooghe, M., Janssens, G., Callebaut, A., and Demoen, B., Abstract Interpretation: Towards the Global Optimization of Prolog Programs, in: *Proc. of the 4th Symp. on Logic Programming*, San Francisco, IEEE Society Press, 1987, pp. 192-204.
- [4] Chase, D. R., Wegman, M., and Zadeck, F. K., Analysis of Pointers and Structures, in: *Proc. of the ACM SIGPLAN'90 Conf. on Programming Language Design and Implementation*, SIGPLAN Notices, 25(6): 296-310 (1990).
- [5] Cousot, P., and Cousot, R., Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints, in: *Proc. of the 4th ACM Symp. on Principles of Programming Languages*, Los Angeles, 1977, pp. 238-252.
- [6] Debray, S. K., Efficient Dataflow Analysis of Logic Programs, in: *Proc. of the 15th ACM Symp. on Principles of Programming Languages*, San Diego, California, 1988, pp. 260-273.
- [7] Foster, I., and Winsborough, W., Copy Avoidance through Compile-Time Analysis and Local Reuse, in: *Proc. of the International Logic Programming Symp.*, MIT Press, 1991.
- [8] Hudak, P. and Bloss, A., The Aggregate Update Problem in Functional Programming Systems, *Conf. Rec. Twelfth ACM Symp. on Principles of Programming Languages*, 1985, pp. 300-314.
- [9] Hudak, P., A Semantic Model for Reference Counting and Its Abstraction, in: S. Abramsky and C. Hankin (eds.), *Abstract Interpretation of Declarative Languages*, Chichester, Ellis Horwood, 1987, pp. 45-62.
- [10] Inoue, K., Seki, H., and Yagi, H., Analysis of Functional Programs to Detect Run-Time Garbage Cells, *ACM Transactions on Programming Languages and Systems*, 10(4): 555-578 (1988).
- [11] Janssens, G., and Bruynooghe, M., Deriving Descriptions of Possible Values of Program Variables by Means of Abstract Interpretation, Report CW107 (to appear in *J. Logic Programming*), Dept. of Comp. Sc., K.U. Leuven, 1990.
- [12] Jones, N. D., and Muchnick, S. S., Flow Analysis and Optimization of LISP-like Structures, in: S. S. Muchnick and N. D. Jones (eds.), *Program Flow Analysis*, Prentice-Hall, 1981, pp. 102-131.
- [13] Jones, N. D., and Søndergaard, H., A Semantic-based Framework for the Abstract Interpretation of Prolog, in: S. Abramsky and C. Hankin (eds.), *Abstract Interpretation of Declarative Languages*, Chichester, Ellis Horwood, 1987, pp. 123-142.
- [14] Kluźniak, F., Compile Time Garbage Collection for Ground Prolog, in: R. A. Kowalski and K. A. Bowen (eds.), *Proc. of the 5th Int. Conf. and Symp. on Logic Programming*, MIT Press, 1988, pp. 1490-1505.
- [15] Larus, J. R., and Hilfinger, P. N., Detecting Conflicts between Structure Accesses, in: *Proc. of the ACM SIGPLAN'88 Conf. on Programming Language Design and Implementation*, SIGPLAN Notices, 23(7): 21-34 (1988).
- [16] Mariën, A., Janssens, G., Mulkers, A., and Bruynooghe, M., The Impact of Abstract Interpretation: An Experiment in Code Generation, in: G. Levi and M. Martelli (eds.), *Proc. of the 6th Int. Conf. on Logic Programming*, MIT Press, 1989, pp. 33-47.
- [17] Marriott, K. and Søndergaard, H., Semantics-Based Dataflow Analysis of Logic Programs, in: G. Ritter (ed.), *Information Processing 89*, N.Y., North-Holland, 1989, pp. 601-606.
- [18] Mellish, C. S., Abstract Interpretation of Prolog Programs, in: S. Abramsky and C. Hankin (eds.), *Abstract Interpretation of Declarative Languages*, Chichester, Ellis Horwood, 1987, pp. 181-198.
- [19] Mulkers, A., Winsborough, W., and Bruynooghe, M., Analysis of Shared Data Structures for Compile-Time Garbage Collection in Logic Programs, in: D.H.D. Warren and P. Szeredi (eds.), *Proc. of the 7th Int. Conf. on Logic Programming*, MIT Press, 1990, pp. 747-762. Also Report CW117 (Extended version), Dept. of Comp. Sc., K.U. Leuven, 1990.
- [20] Mulkers, A., Deriving Live Data Structures in Logic Programs by means of Abstract Interpretation, Ph.D. thesis, Dept. of Comp. Sc., K.U. Leuven, 1991.
- [21] Van Caneghem, M., L'anatomie de Prolog, InterEditions, 1986.
- [22] Winsborough, W., Path-Dependent Reachability Analysis for Multiple Specialization, in: E. Lusk and R. Overbeek (eds.), *Proc. of the North American Conf. on Logic Programming*, MIT Press, 1989, pp. 133-153, (full paper to appear in *J. Logic Programming*).

PROCEEDINGS

Computer Systems and Software Engineering

6th Annual
European
Computer
Conference

Sponsored by

IEEE
Region 8
Benelux Section

IEEE Computer
Society

Delft University of
Technology

TU Delft

COMP
EURO 92

The Hague, May 4-8, 1992

Netherlands Congress Centre

Edited by Patrick Dewilde and Joos Vandewalle