Incremental Flow Analysis

Andreas Krall and Thomas Berger
Institut für Computersprachen
Technische Universität Wien
Argentinierstraße 8
A-1040 Wien
{andi,tb}@mips.complang.tuwien.ac.at

Abstract

Abstract interpretation is a technique for flow analysis widely used in the area of logic programming. Until now it has been used only for the global compilation of Prolog programs. But the programming language Prolog enables the programmer to change the program during run time using the built-in predicates `assert` and `retract`. To support the generation of efficient code for programs using these dynamic database predicates we extended abstract interpretation to be executed incrementally. The aim of incremental abstract interpretation is to gather program information with minimal reinterpretation at a program change. In this paper we describe the implementation of incremental abstract interpretation and the integration in our VAM_{1P} based Prolog compiler. Preliminary results show that incremental global compilation can achieve the same optimization quality as global compilation with little additional cost.

1 Introduction

Since several years we do research in the area of implementation of logic programming languages. To support fast interpretation and compilation of Prolog programs we developed the VAM_{2P} (Vienna Abstract Machine with two instruction pointers) [Kra87] [KN90] and the VAM_{1P} (Vienna Abstract Machine with one instruction pointer) [KB92]. Although the VAM manages more information about a program than the WAM (Warren Abstract Machine) [War83], it is necessary to add flow analysis for type inference and reference chain length calculation. Extending our global VAM_{1P} compiler by support for dynamic database predicates it became apparent that the data structures necessary for storing the information for dynamic compilation were sufficient to integrate incremental global abstract interpretation.

In chapter 2 we present our approach to global data flow analysis and show how it can be solved incrementally. Chapter 3 contains details of our prototype implementation and chapter 4 presents some preliminary results.

2 Abstract Interpretation

Abstract interpretation is a technique of describing and implementing global flow analysis of programs. It was introduced by [CC77] for data flow analysis of imperative languages.

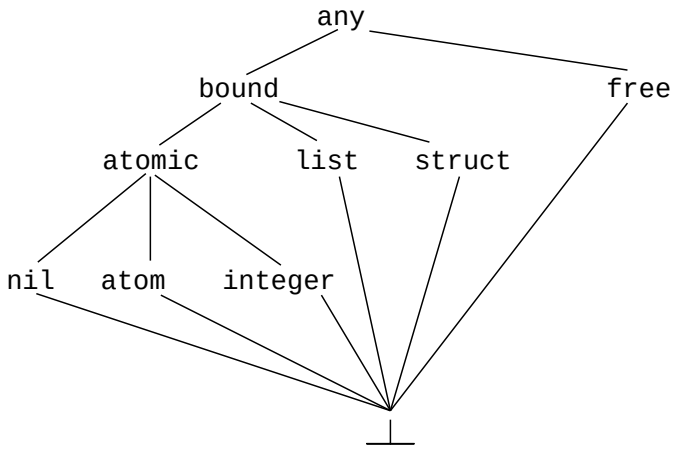

Figure 1: Abstract domain for type and mode inference

This work was the basis of much of the recent work in the field of logic programming [AH87] [Bru91] [Deb92] [Mel85] [RD92] [Tay89]. Abstract interpretation executes programs over an abstract domain. Recursion is handled by computing fix points. To guarantee the termination and completeness of the execution a suitable choice of the abstract domain is necessary. Usually the elements of this domain form a lattice.

2.1 Global Abstract Interpretation

We do gather information about mode, type and reference chain length of variables. Trailing information is not handled by the abstract interpreter because the VAM does trail more seldom than the WAM. Fig. 1 describes the lattice of the abstract domain types. Each type represents a set of terms:

any	is the top element of the domain
free	describes an unbound variable and contains a reference to all aliased variables
bound	describes non-variable terms
atomic	is the supertype of nil, atom and integer
list	is a non empty list (it contains the types of its arguments)
struct	is a structure (information about the functor and arguments is contained)
nil	represents the empty list
atom	is the set of all atoms
integer	is the set of all integer numbers

Possible infinite nesting of compound terms makes the handling of the types list and struct difficult. Therefore, interpretation stops after the allowed limit of nested compound terms is reached. To gather useful information about recursive data structures a recursive type is introduced which contains also the information about the termination type. The type *list(int,rec(nil))* describes the type list of integer.

Each type of the domain additionally contains information about the reference chain length. Valid values for this length are 0, 1 and unknown. A length of e.g. 0 will be extracted during the unification of a new variable (the first occurrence) and a constant

term, 1 is typical for the unification of two different new variables (the compiler generates the macro firstvar_firstvar, which generates a free cell on the copy stack and makes both variables a reference to that cell). If the length of the reference chain is not decidable, its length information gets the value unknown.

We use a top-down approach for the analysis to extract the desired information. Different calls to the same clause are handled separately to get the exact types. The gathered information is a description of the living variables. Recursive calls of a clause are computed until a fix point for the gathered information is reached. If a call is computed the second time there exists already information about this call. Now the old information is compared with the new one and the interpretation stops for this clause at this point, if the new information is more special, i.e. the union of all the old types and the new types is equal to the old types. The gathered information has reached a fix point.

2.2 Incremental Abstract Interpretation

In most Prolog systems the modification of the database leads to a mixture of interpreted and compiled code. Optimization information is not available for dynamically asserted clauses. In some systems it is not allowed to assert clauses for a predicate which is not declared to be dynamic. We wanted to support dynamic database predicates without any loss of optimization information. The idea of incremental abstract interpretation is to get this information with minimal reinterpretation. Changing the database by asserting or retracting clauses causes a local analysis of all procedures whose variable domains are changed. In the worst case the assertion of a new clause can take effect in the abstract reinterpretation and recompilation of the whole program.

To support incremental abstract interpretation it is necessary to store additional information with the compiled code in the code area. This information consists of:

- the intermediate code of all clauses
- description of variables and their domains
- table containing the entry points for the clause codes
- table containing the callers (parents) of a clause

Incremental abstract interpretation starts the local analysis with all the callers of the modified procedure and interpret the intermediate code of all dependent procedures. Interpretation is stopped if the new domains are equal to the domains derived by the previous analysis. If the domain has not changed, new code is generated only for the changed part of the program. If the domain has been changed and the new domain is a subtype of the old domain, the previously generated code fits for the changed program part. The information derived for the program before the change is less exact than the possibly derivable information. Incremental interpretation can stop now. If the new domain is instead a supertype of the old one, the assertion of the new clause made the possible information less precise and the old code wrong for the new program. Therefore, incremental abstract interpretation must further analyse the callers of the clause.

The retraction of a clause has a similar effect as the assertion of a new one. The only difference is that there cannot be an information loss which makes the previously generated

code of the callers of the calling clauses wrong. It is sufficient to start the incremental analysis with the calling clauses of the changed procedure and interpret top-down until the derived information is equal to the previously inferred one.

3 Implementation

Our prototype compiler is implemented in Prolog. To make incremental abstract interpretation possible some extensions to the Prolog interpreter were necessary. We implemented two new builtin-predicates. One assembles a list of symbolic assembly language instructions (produced by the compiler) and saves the machine code in the code area. The second builtin calls an assembled program and after completion returns to the interpreter.

The compilation process comprise the following steps:

- read a clause and compile it into intermediate code
- incrementally execute abstract interpretation
- compile all changed clauses to VAM_{1P} code
- transform the VAM_{1P} instructions into assembly language instructions
- perform instruction scheduling
- assemble the instructions and save the instructions in the code area
- update branch instructions (connect the new code to the older one)

Our intermediate code is based on VAM_{2P} code enriched by information describing the abstract domain of variables. This approach is similar to the compiling data flow analysis of [TL92] with the difference, that our intermediate code is not based on the WAM and our abstract VAM_{2P} interpreter is implemented in Prolog. The gathered information is used for:

- reducing the code for dereferencing variables,
- eliminating tag checkings,
- eliminating the unification code or the copy code handling structure arguments,
- eliminating parts of the code for clause indexing
- eliminating unnecessary choice points

To perform the incremental interpretation some information about the compiled program has to be available. The intermediate code is stored in the database at run time so that it can be used by the incremental interpretation. For the entry points of all possible calls to all clauses the following information is stored in a table:

- the start address of the compiled code in memory
- the addresses of all start points of the procedures
- the addresses of all branches to procedures

Recompilation after incremental analysis assembles the new produced code to a free part in the code area and modifies the branches of the unchanged code to branch to the start address of the called recompiled clauses.

benchmark	run time		compile time	
	not optimized ms	optimized scaled	static ms	incremental scaled
det.append	0.0096	1.12	12933	2.8
naive reverse	0.2010	1.13	14287	3.2
quicksort	0.8073	1.24	19883	4.2
8-queens	9.5894	1.08	10826	3.1
serialize	0.6732	1.09	22219	3.5
differentiate	0.1655	1.19	34081	4.8
query	5.6349	1.39	19754	4.9
bucket	49.203	1.22	20712	3.7
permutation	790.97	1.29	15931	2.1

Table 1: run time and compilation time for the benchmarks

4 Results

Preliminary results show that incremental abstract interpretation can collect the same information as global abstract information.

We compiled a set of small benchmarks on a DecStation 5000/200 (25MHz MIPS R3000 processor). A description and the source code of the benchmarks can be found in [Bee89]. All benchmarks were compiled without abstract interpretation, with static abstract interpretation and incrementally with abstract interpretation (see Tab. 1).

Computation times were measured by executing each benchmark between 100 and 30000 times. Tab. 1 shows that abstract interpretation made possible a speedup between 10 and 40 percent. E.g. the query benchmark executes 1.39 times faster than without abstract interpretation.

Tab. 1 compares also the compilation times. Incremental compilation needs between two and five times as much time as static compilation for the benchmarks. For incremental compilation each clause of the program was read by the compiler, compiled to intermediate code, incrementally interpreted over the abstract domain, all already compiled clauses of the program whose variable domains were changed through the new clause were recompiled, the code scheduled and assembled into memory. The new intermediate code and the table containing the branch addresses were asserted as a fact to the database.

Tab. 2 compares the code sizes produced by the incremental and the static methods. Due to the reduction of dereferencing and tag checking and refined constant indexing the optimized programs needs only 64 to 92 percent of the size of the unoptimized one. Incremental compilation increases the sizes because we have no garbage collector for the code area where the incrementally compiled code is placed.

5 Further Work

The prototype implementation has demonstrated that it is feasible to support abstract interpretation for Prolog programs using dynamic database predicates. To enhance the

benchmark	size		
	not optimized byte	static scaled	incremental scaled
det.append	9864	0.92	1.2
naive reverse	8536	0.91	1.8
quicksort	12792	0.88	1.1
8-queens	9744	0.76	1.2
serialize	16232	0.87	1.8
differentiate	40320	0.76	2.9
query	14128	0.64	2.5
bucket	15988	0.69	2.3
permutation	8948	0.84	1.7

Table 2: sizes of the compiled programs

speed of our incremental compiler and to reduce the size of the data structures we will implement the incremental compiler in C. The incremental abstract interpreter will execute modified VAM_{2P} code. The integration of the incremental compiler in the run time environment enables the execution of interpreted VAM_{2P} code and incrementally compiled VAM_{1P} code.

Acknowledgement

We express our thanks to Anton Ertl, Ulrich Neumerkel and Franz Puntigam for their comments on earlier drafts of this paper.

References

[AH87] Samson Abramsky and Chris Hankin, editors. *Abstract Interpretation of Declarative Languages*. Ellis Horwood, 1987.

[Bee89] Joachim Beer. *Concepts, Design, and Performance Analysis of a Parallel Prolog Machine*. Springer, 1989.

[Bru91] Maurice Bruynooghe. A practical framework for the abstract interpretation of logic programs. *Journal of Logic programming*, 10(1), 1991.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Fourth Symp. Priciples of Programming Languages*. ACM, 1977.

[Deb92] Saumya Debray. A simple code improvement scheme for Prolog. *Journal of Logic Programming*, 13(1), 1992.

[KB92] Andreas Krall and Thomas Berger. Fast Prolog with a VAM_{1P} based Prolog compiler. In *PLILP'92*, LNCS. Springer 631, 1992.

[KN90] Andreas Krall and Ulrich Neumerkel. The Vienna abstract machine. In *PLILP'90*, LNCS. Springer, 1990.

[Kra87] Andreas Krall. Implementation of a high-speed Prolog interpreter. In *Conf. on Interpreters and Interpretative Techniques*, volume 22(7) of *SIGPLAN*. ACM, 1987.

[Mel85] Christopher S. Mellish. Some global optimizations for a Prolog compiler. *Journal of Logic Programming*, 2(1), 1985.

[RD92] Peter Van Roy and Alvin M. Despain. High-performance logic programming with the Aquarius Prolog compiler. *IEEE Computer*, 25(1), 1992.

[Tay89] Andrew Taylor. Removal of dereferencing and trailing in Prolog compilation. In *Sixth International Conference on Logic Programming*, Lisbon, 1989.

[TL92] Jichang Tan and I-Peng Lin. Compiling dataflow analysis of logic programs. In *Conference on Programming Language Design and Implementation*, volume 27(7) of *SIGPLAN*. ACM, 1992.

[War83] David H.D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, 1983.