Abstract Interpretation of PIC programs through Logic Programming

Henriksen, Kim Steen; Gallagher, John Patrick

Published in:
Sixth IEEE International Workshop on Source Code Analysis and Manipulation

Publication date:
2006

Document Version
Publisher's PDF, also known as Version of record

Citation for published version (APA):
Henriksen, K. S., & Gallagher, J. P. (2006). Abstract Interpretation of PIC programs through Logic Programming. In *Sixth IEEE International Workshop on Source Code Analysis and Manipulation* (pp. 184-193). IEEE.

Abstract Interpretation of PIC programs through Logic Programming

Kim S. Henriksen
Computer Science
Roskilde University
Roskilde, Denmark
kimsh@ruc.dk

John P. Gallagher
Computer Science
Roskilde University
Roskilde, Denmark
jpg@ruc.dk

Abstract

A logic based general approach to abstract interpretation of low-level machine programs is reported. It is based on modelling the behavior of the machine as a logic program. General purpose program analysis and transformation of logic programs, such as partial evaluation and convex hull analysis, are applied to the logic based model of the machine.

A small PIC microcontroller is used as a case study. An emulator for this microcontroller is written in Prolog, and standard programming transformations and analysis techniques are used to specialise this emulator with respect to a given PIC program. The specialised emulator can now be further analysed to gain insight into the given program for the PIC microcontroller.

The method describes a general framework for applying abstractions, illustrated here by linear constraints and convex hull analysis, to logic programs. Using these techniques on the specialised PIC emulator, it is possible to obtain constraints on and linear relations between data registers, enabling detection of for instance overflows, branch conditions and so on.

1 Introduction

In this article we demonstrate source code analysis based on an implementation of the semantics of the language to be analysed, in logic programming. Applying general analysis techniques, programming transformation tools etc. for logic programs, it is possible to obtain e.g. liveness analysis results and perform a convex hull analysis.

We have used a small PIC microprocessor as a case-study. However, the method described is not specific to a particular language or abstract machine. It can easily be applied to other languages or abstract machines in the same manner as described here.

The method is not efficient in terms of resources needed to perform the analysis; but it is a very general approach, where well documented program analysis and transformation techniques and existing program transformation tools are applied to an emulator of the language or abstract machine. Once the emulator has been implemented, the following specialisation steps requires little or no tweaking to produce analysis results.

Using logic programming as the language for analysing the PIC processor has a few advantages. Analysis and transformation tools are readily available and it is a language well suited for both. Reasoning about or even proving correctness of methods and results are also easier to do for a high level language, such as Prolog.

The analysis consists of five parts:

- an emulator is written in Prolog for the language or abstract machine
- a control flow analysis is performed of the given program to be analysed
- the emulator is partially evaluated wrt. the given program and the results of the control flow analysis
- program transformation tools can now be applied to the specialised emulator to gain e.g. liveness analysis results
- a bottom-up evaluation of the partially evaluated emulator uses the Parma Polyhedra Library to solve constraints on data memory

The result will be a set of constraints on the live data registers, forming a convex hull, for each program point in the program to be analysed. For the PIC micro-controller that we use as a case-study, software is developed in assembly language. Debugging can be difficult since the only interface to the micro-controller are the I/O ports. The constraints can serve as an aid to programmer to determine under which conditions a given register will overflow, reach zero and so on.

A web-based interface for the analysis tools has been built, that will allow a PIC programmer to upload a program and have the constraints presented along the with program. The application of the specialisation and transformation tools can all be done automatically. The PIC programmer need not see any specialised logic programs.

2 Emulating an abstract machine

We have previously used the procedure described in this section for analysing read/write patterns of data memory, dead code etc. of abstract machines [14]. The procedure followed here differs on the representation of the machine state and the initial Control Flow Analysis.

As an example abstract machine to work with, we have chosen the PIC microcontroller. It is a very small device, used in e.g. wearable computing. One of its advantages is its low power consumption. We have chosen a particular model number to emulate, namely the PIC16F84, which has 35 single word instructions, 1024 words of program memory, 68 bytes of data memory, an eight-level deep hardware stack and two I/O ports.

The emulator is implemented in Prolog, and a predicate called `execute` contains the main loop that executes each instruction.

```
execute(Prog,State,Env) :-
  fetchInst(Prog,State,Instr,Arg1,Arg2),
  execInst(Instr,Arg1,Arg2,State,State1),
  simulateEnv(State1,StateOut,Env),
  execute(Prog,StateOut,Env).
```

A state is a grouping of lists and values of the form `state(Regs,PC,Acc,Stack)`. The environment can be used to simulate external input to the processor. The argument *Regs* contains the state of the data memory. It is a list of the form `[RegisterNumber - Value]`.

There exists an `execInst`-clause for every machine instruction in the PIC instruction set.

Example 1 *As an example the* `addwf` *instruction that performs an add operation between the accumulator and a register, storing the result back into the accumulator, would be implemented as shown.*

```
execInst(addwf,Arg1,0,
         state(RegIn,Stack,PC,Acc),
         state(RegOut,Stack,PCOut,AccOut)) :-
  retrievedata(PC,RegIn,Ru1,Arg1,X),
  intAdd(Wt,X,Acc), reduceBits(Wt,AccOut),
  intShiftR(C,Wt,8), updatez(PC,Ru1,Rt,AccOut),
  updatec(PC,Rt,RegOut,C), PCOut is PC + 1.
```

The implementation is not efficient but it is generic. The data stored in a state and the instruction set can easily be changed to emulate other processors or microcontrollers.

3 Partial Evaluation of Emulator

The emulator described in Section 2 is specialised with respect to a given program, and a set of Control-Flow facts generated from a Control Flow Analysis of the PIC program - these are facts of the form `nextInstr(PC,PCnext)`. In Section 5 the Control Flow Analysis, and how these `nextInstr/2` facts are derived, is described in detail.

For the specialisation step we use an off-line partial evaluator for Prolog, called Logen [18]. The PIC program and any environment data supplied are static inputs. In the `execute`-loop, everything is unfolded except the loop itself. Every `execInst` is unfolded completely, only in the integer domain operations we memo those operators calculating the (new) values in the machine state. The later Convex Hull Analyser step will be based on those memo'ed operations.

The `return`-instruction is unfolded wrt. to the `nextInstr/2`-facts. The stack content will not be known at specialisation time, but the unfolding of the `nextInstr/2`-fact will ensure that control flow is returned to any possible instruction that could occur at the top of the stack, at that particular `return`-instruction. The implementation of the `return`-instruction is shown below.

```
execInst(return,_,_,
         state(RegIn,StackIn,PC,Acc),
         state(RegIn,StackOut,PCOut,Acc)) :-
  popstack(StackIn,StackOut,PCOut),
  nextInstr(PC,PCOut).
```

The result of this specialisation is a new Prolog program, with a numbered `execute`-predicate for every program point in the original PIC program. The control flow of the PIC program is now embedded in the calls from one execute clause to the next. PIC instructions that can alter control flow of the program will have more than version of a given numbered `execute`-clause. Instructions modifying the hardware status register (containing among other things, the carry and zero bit) will also have more than one version, depending on wether the results would set or not set a bit in the status register.

Example 2 *An example of such an* `execute`*-clause after specialisation is shown below; this particular clause corresponds to the* `addwf` *instruction whose* `execInst`*-implementation was shown earlier in Example 1.*

```
execute__5(S,Q,O,M,K,I,G,E,D,B,C,_,F,H,P,A) :-
  T is Q+A, 0 is T>>8,
  T \== 0, _ is 24 /\ 251,
  U is 24 /\ 254, _ is _+1,
  execute__6(S,Q,O,M,K,I,G,E,D,B,C,U,F,H,P,T).
```

Two status bits can be modified, so 4 different versions are generated; control flow of the program can only pass on to the following instruction, so all `execute__5`s will continue with `execute__6`.

4 Machine State Liveness

The specialised emulator resulting from the partial evaluation described in previous sections is an automatically generated program. Such automatically generated programs often contains redundant parts, that can be eliminated by general purpose program transformations without affecting the correctness of the program. The clause shown in Example 2 contains various redundancies including a large number of arguments for the predicate `execute_5` and some redundant operations in the clause body.

After specialisation of the emulator, the semantics of the PIC program is now embedded in the `execute`-clauses of the specialised emulator. The machine state is embedded into the arguments of the `execute`-clauses. To simplify later specialisation and analysis steps, the machine state can be reduced to only the live state. Eliminating dead elements of the state, will not alter the semantics of the program.

Two approaches to liveness analysis of PIC programs have been considered: a method based on Datalog program properties (see Section 5.2) and a general program transformation technique for logic programs. We describe the latter approach in this section.

A liveness analysis would typically consist of

- a flow graph
- an annotation of the flow graph along the edges, with *reference* and *definition* statements, where *reference* is the use of a data element and *definition* is the assignment of a data element.
- a method based on the flow graph and its annotation, for solving which data elements contain live values at which program points.

4.1 Liveness Analysis Using Redundant Argument Filtering

Leuschel and Sørensen [19] proposed a general logic program transformation called "redundant argument filtering". This transformation removes predicate arguments that are never "used". There are two forms of the transformation, corresponding to top-down and bottom-up propagation of information. The main motivation for the transformation was the simplification of programs produced by other transformations, in particular by conjunctive partial deduction [10]. We focus here on the transformation called FAR (Section 5 of [19]). In this section we show how the FAR algorithm is a generalisation of liveness analysis and can be applied to the specialised PIC emulator to eliminate dead registers at each program point. The result is a simplified program for program analysers.

4.1.1 Correct Erasures

The FAR algorithm computes a correct *erasure* for a given logic program P; an erasure is a set of predicate argument positions that can be eliminated without affecting the computed answers for any goal. More precisely, let an erasure E be a set of predicate arguments (p, j) that are to be removed. For example suppose an erasure contains $\{(p,1),(p,3),(p,4)\}$. Then this means that the first, third and fourth arguments of predicate p are to be removed. Given a program P and erasure E, denote by $P|E$ the result of striking out the arguments in E from all occurrences of the predicates in P. In the case of the erasure $\{(p,1),(p,3),(p,4)\}$ an occurrence of an atomic formula for p such as $p(t_1,t_2,t_3,t_4,t_5)$, would be replaced by $p(t_2,t_5)$.

An erasure E is *correct* for program P if the following property holds. For every computation of P with a goal G, the answers and finite failures are the same (for the non-erased arguments) as the computation of $P|E$ with $G|E$ (the result of applying the erasure to G).

4.1.2 Algorithm for Computing a Correct Erasure (FAR)

The algorithm presented by Leuschel and Sørensen successively identifies arguments that are "needed". Initially, the erasure for P is the set of all argument positions of predicates in P. On each iteration, arguments are removed from the erasure until a correct erasure is computed.

```
FAR Algorithm Input:  program P.
Initialise
   i = 0;
   E_0 = the set of all predicate arguments;
while (there exists a (p,k) ∈ E_i
   and a clause p(t_1,...,t_n) ← B in P
   such that
      1.  t_k is not a variable; or
      2.  t_k is a variable occurring more
          than once in p(t_1,...,t_n); or
      3.  t_k is a variable occurring in B|E_i
do E_{i+1} = E_i \ {(p,k)};  i = i + 1;
return E_i
```

It is provable that if conditions 1-3 checked in the loop are false for all argument positions then the corresponding erasure is correct.

4.1.3 Liveness Analysis using the FAR algorithm

We now show that the FAR algorithm can be used as a liveness analyser. In fact it is more general than a liveness analyser; as will be seen, it can remove redundancies other than dead variables. To do this we build a straightforward translation from control-flow graphs and logic programs called control-flow programs. Then we show that the classical liveness analysis on control-flow graphs [1] is mimicked by the action of the FAR algorithm on control-flow programs.

Note that the flow-programs and flow-graphs considered in this section are not part of the PIC case study; they are just defined in order to show that the FAR algorithm can perform liveness analysis.

We take flow-graphs to consist of a set of vertices (program points) and directed edges labelled either by an assignment statement $x = e$ or a boolean expressions b. Conditional branches are represented by edges labelled with a boolean expression, and control flows along that edge if the condition evaluates to $true$. Control flow graphs with branch instructions and goto statements could be defined instead, without altering anything essential in the procedure below.

Let $x_1, \ldots, x_m$ be the set of all variables appearing in the graph (i.e. in assignment statements or in boolean expressions). For each vertex j define a unique predicate p_j with m arguments. The control-flow program resulting from a given graph is defined to be the set of clauses of the following form.

1. $p_i(x_1, \ldots, x_m) \leftarrow x'_l = e, p_j(x_1, \ldots, x'_l, \ldots x_m)$, where $x_l = e$ is an assignment statement on the edge (i, j) and x'_l is a variable different from $x_1, \ldots, x_m$.
2. $p_i(x_1, \ldots, x_m) \leftarrow b, p_j(x_1, \ldots, x_m)$, where b is a boolean expression on the edge (i, j).

Let 0 be the entry vertex of the flow-graph. We observe that there is a computation in the flow-program that calls some sequence of predicates $p_0, p_{i_1}, p_{i_2}, p_{i_3}, \ldots$ iff there is a legal computation in the flow-graph passing through vertices $0, i_1, i_2, i_3, \ldots$. The arguments of a call to predicate p_i represent the state of the variables in the program at vertex i in the corresponding flow-graph execution.

Now consider the application of the FAR algorithm to this program. Initially, E_0 contains every pair (p_j, k) where j is a vertex and $1 \leq k \leq m$. (We assume that the arguments of the equality predicate and the boolean expression predicates that appear in the program are not erased.) We argue informally that the iterations of the FAR algorithm acting on the flow-program correspond to the iterations of the classical liveness algorithm. Consider an edge (i, j) labelled by an assignment statement $x_l = e$. Define, as is usual, $use(i) = vars(e)$ and $def(i) = \{x_l\}$. If the edge is labelled by a boolean expression b then $use(i) = vars(b)$ and $def(i) = \{\}$.

Consider some erasure; let the set of arguments that are not erased from p_i be called $in(i)$. Let the set of arguments that are not erased from all predicates p_j such that there is and edge (i, j) be called $out(i)$. Then we can show that the FAR algorithm solves the classic dataflow equations for liveness, namely $in(i) = use(i) \cup (out(i) - def(i))$, and $out(i) = \bigcup_{j \in succ(i)} in(j)$, where $succ(i)$ is the set of j such that there is an edge (i, j). The algorithm iterates starting from initial values $in(i) = out(i) = \{\}$ in the classic liveness analysis; this corresponds to the fact that the initial erasure consists of all predicate arguments. When the FAR algorithm terminates, the sets $in(i)$ contain the set of arguments that are *not* erased from predicate p_i, that is, the live variables at vertex i. The detailed proof argues that the FAR loop terminates exactly when the above dataflow equations are satisified.

Claim 1 *Given a control flow-graph containing variables $x_1, \ldots, x_m$ and the corresponding control-flow program as defined above. Then according to the classical liveness analysis, a variable x_j is live at a given node i of the control flow graph ($x_j \in in(i)$) iff the FAR algorithm returns an erasure that does not include (p_i, j).*

4.1.4 Application of Redundant Argument Filtering

The specialised emulator described in Section 3 has a similar structure to the flow-programs described above. The state of the registers is held in the arguments of the `execute` predicates, and there is one version of the `execute` predicate for each PIC program point. Application of the FAR algorithm to the specialised emulator results in the erasure of all register arguments from say `execute`$_i$ that are dead at the program point corresponding to `execute`$_i$. The number of registers live at a point is often a small fraction of the total set of registers, and hence the FAR algorithm yields a program that can be much more efficient to analyse, without losing any essential information about the program.

In fact, redundant argument filtering does more than just removing dead arguments. It also allows some operations in the body of the `execute` clauses to be eliminated as they definitely succeed. Example 3 shows an example of the clause for the same predicate `execute_5` shown in Example 2. Note that the number of arguments of `execute_5` has been drastically reduced and the clause body is simpler.

Example 3 *An example of an* `execute`*-clause after specialisation and redundant argument filtering*

```
execute__5(B,A) :-
  C is B+A, 0 is C>>8,
  C \== 0, D is 24 /\ 254,
  execute__6(D,B,C).
```

5 Flow Properties Expressed in Datalog

It was mentioned earlier that the specialisation of `return` instructions in the emulator requires the addition to the emulator of a set of facts of the form `nextInstr(X,Y)`. In PIC programs, there is no syntactic structure identifying procedures; when a subroutine is invoked by a `call`-instruction, the program point following the `call`-instruction is pushed onto a stack. Upon exit

from the subroutine by a `return`-instruction, the stack is popped and control is returned to the program point at the top of the stack. There may be several calls to the same subroutine at different program points. Thus calls and returns are matched dynamically and some flow analysis is required in order to generate the possible call points matching a given return.

In this section we show how to generate a set of facts `nextInstr(X,Y)` such that for a given return point `X` the set of facts `nextInstr(X,Y)` shows at least all the possible points to which control can continue. Note that the set of points `Y` such that `nextInstr(X,Y)` is in general an over-approximation of the set of values that can appear at runtime at the top of call stack when return instruction `X` is reached. We could of course easily generate the facts `nextInstr(R,C)` where `R` is any return program point and `C` is any call program point. This would be safe but would in general lead to much larger specialised programs and preclude the detection of some cases of dead code. The method described below gives a more precise result.

5.1 Datalog as a property modelling language

Datalog [21] is a logic programming language in which there are no function symbols with arity greater than zero. Efficient techniques for computing Datalog models have been studied extensively in research on deductive database systems. Recently, Datalog has been applied successfully in program analysis tasks. The approach is simple: express some properties of interest as Datalog predicates, provide Datalog rules and facts (that is, a Datalog program) defining their intended interrelationships, and then compute the least model of the complete Datalog program to obtain an explicit listing of the properties as a set of facts. Note that least models of Datalog programs are finite and that negations in Datalog programs can be handled provided that they are *stratified*. An efficient BDD-based toolset for computing Datalog program models is available [23], and it has been applied to Java programs containing thousands of lines [22, 17]. In previous work we have also used Datalog programs as abstractions of full logic programs [13] and in that work the same *bddbddb*-package was used.

5.1.1 Datalog rules for PIC control flow

We now present a Datalog program representing the control flow of a PIC program. Each instruction of the machine is given a unique number, and for each instruction, we annotate it with a set of facts that describe the behaviour of that particular instruction. Note that this is done once and for all for a given machine language, not for each user program.

For the control flow analysis, a single fact for each instruction in the PICs instruction set will suffice, so we only need to categorise each instruction depending on its modification of the control flow of the program. For the PIC there are five different types of instructions:

- Straight line instruction: Upon execution of such an instruction, the control flow of the program continues with the next instruction in the program.
- Branch instruction: When a branch instruction is executed, either the following instruction in the program is executed or it will be skipped and the second following instruction will be executed.
- Goto instruction: Control flow of the program will continue with the instruction number given as an argument to the `goto`-instruction.
- Call instruction: Same as a goto-instruction, but it is distinguished from the gotos since the matching return instruction can redirect control flow of the program, to the instruction following the call.
- Return instruction: The instruction to be executed after a return instruction will depend on the call context of that return instrucion.

All programs are assumed to start at program point 0, so we add that as a fact as well - `entryPoint(0).`

The program is encoded as a set of facts of the form `pp(PC,I,A1,A2)`, where `PC` is the program memory location of that instruction, `I` is the instruction code and `A1` and `A2` are the arguments to that instruction.

As Datalog does not allow for arithmetic expressions, a simple increment predicate will be need for the Control Flow Analysis. A set of increment facts are added to the Datalog program. The increment operator is defined as `increment(0,1). increment(1,2).` etc. up to the maximum number of instructions in the program.

Once every instruction has been annotated with the facts described above, and the PIC program has been appended, rules can be written to detect properties of the associated program.

Example 4 *Program point* `N` *is a branch instruction if the instruction located in the program memory at that address is a branch instruction. This translates into the rule*

```
branchInstr(N) :- pp(N,I,_,_),
branchInst(I).
```

A *next instruction* procedure can be defined in terms of the facts from the control flow list. For each type of fact, a rule like the one shown in Example 4 is created. The *next instruction* procedure has a rule for each type of fact.

```
nextInstr(N1,N2) :- straightLineInstr(N1),
  increment(N1,N2).
nextInstr(N1,N2) :- branchInstr(N1),
  increment(N1,N2).
nextInstr(N1,N3) :- branchInstr(N1),
  increment(N1,N2), increment(N2,N3).
nextInstr(N1,R1) :- gotoInstr(N1,R1).
nextInstr(N1,R1) :- callInstr(N1,R1).
nextInstr(N1,N3) :- returnInstr(N1),
  calledFrom(N1,N2), increment(N2,N3).
```

The branch instruction has two rules. Conditions for branch instructions are not taken into account, and control flow can therefore either continue with the following or the second following instruction. Depending on which abstract machine is being modeled, e.g. branch instructions can behave differently.

Note that the next instruction following a return instruction is defined using a predicate `calledFrom`. The `calledFrom` procedure determines the *call context*s for a given instruction. A *call context* corresponds to the latest `call` instruction that was encountered in the control flow of the program. The `calledFrom` procedure is defined in terms of the facts from the control flow category and the following rules:

```
calledFrom(0,0).
calledFrom(N1,N2) :- increment(N3,N1),
  straightLineInstr(N3), calledFrom(N3,N2).
calledFrom(N1,N2) :- increment(N3,N1),
  branchInstr(N3), calledFrom(N3,N2).
calledFrom(N1,N2) :- increment(N3,N1),
  increment(N4,N3), branchInstr(N4),
  calledFrom(N4,N2).
calledFrom(N1,N2) :- gotoInstr(N3,N1),
  calledFrom(N3,N2).
calledFrom(R1,N1) :- callInstr(N1,R1).
calledFrom(N1,N2) :- increment(N3,N1),
  callInstr(N3,_), calledFrom(N3,N2).
```

All programs are assumed to start at instruction at memory location 0, so initially the call context is 0. Since a subroutine can be called from more than one place in the program, an instruction can belong to more than one call context.

The clauses above are augmented with the facts defining a particular program (the `pp(N,I,A1,A2)` facts). The model of this program is computed and the resulting `nextInstr(_,_)` facts represent the edges of the program's flow-graph. The facts are inserted into the emulator before specialising, as explained in Section 3.

5.1.2 Dead code detection

Dead code are instructions in the program that will never be executed. A more optimal use of the program memory can be achieved by eliminating these instructions.

An instruction is dead if it is not reachable from any trace in the program starting at program point 0. This can be expressed by the following Datalog rules:

```
reachable(0).
reachable(N1) :- nextInstr(N2,N1),
  reachable(N2).
unreachable(N) :- pp(N,_,_,_),
  !reachable(N).
```

We introduce the exclamation mark as negation. The rule states that reachable instructions are those that can be traced back to program point 0, and the unreachable instructions are any instruction in the program that are not reachable.

Using this method on program examples provided by the Wearable Computing Lab. at University of Bristol[1], we found 2 dead instructions in a hand optimised program of 400 instructions (arising from over-defensive programming!) and 56 dead instructions in a 231-instruction program (apparently arising from partially redundant code that had been ported from another application).

5.2 Liveness Analysis using Datalog

A similar approach can be used to compute the set of registers live at a given program point. Space does not allow a detailed description. However, in ongoing work we are showing how properties expressible in standard monotonic dataflow frameworks can be systematically expressed and solved in Datalog. This provides a flexible and easily implemented approach to many flow analysis problems.

6 Convex Hull Analysis

The program resulting from the specialisation of the emulator as described up until now, is a logic program equivalent to the initially supplied PIC program. Existing analysis tools and techniques for logic programs can now be applied to the specialised emulator, to reason about the PIC program - e.g. CiaoPP [15], a global program analysis, source to source transformation and optimisation tool for Logic Programs.

In this section we describe a method for applying a particular numerical analysis method, Convex Hull Analysis, to the specialised emulator. We have developed our own tool for this purpose to accommodate the particular boolean operations, like `AND`, `OR` and `NOT`, that are found in the specialised emulator. A convex hull analysis is a numerical abstraction of the variables in a program. The abstraction is a set of constraints and relations between the variables. Polyhedral Convex Hulls, first applied in program analysis by Cousot and Halbwachs [9], have been used for a variety of purposes in program analysis , including in the field of logic and constraint logic programming [3, 4], e.g. for argument-size analysis, time-complexity analysis and termination analysis [16].

[1] `http://wearables.cs.bris.ac.uk/`

Parma Polyhedra Library For a convex hull analyser a few polyhedra operations are required. These are projection, emptiness checking, inclusion testing and convex hulls. The Parma Polyhedra Library (PPL) is a programming library targeted especially at analysis and verification [2]. It implements the operations needed for a convex hull analysis and it has interfaces for a variety of programming languages including Ciao Prolog [6].

6.1 Bottom up analysis

Our analyser is based on a bottom up evaluator for logic programs, developed by Michael Codish [7]. Prolog programs are evaluated top down, but bottom-up analysis computing the least model of the program provides sound information about the set of all possible answers obtained in top-down computations. Additionally, using standard transformations such as "magic-set" transforms and other query-answer transforms, bottom-up evaluation can provide accurate information about the calls that arise in a particular top-down computation.

The naive bottom up interpreter is a small Prolog program, where each clause, $h \leftarrow b_1, ..., b_n$ in the program P to be analysed, is represented as facts of the form $user\_clause(h, [b_1, ..., b_n])$. The program is evaluated iteratively until a fix point is reached. In each iteration, heads whose body can be proven from existing facts, are asserted as new facts themselves. A fixpoint is reached, when no new heads can be asserted.

For the Convex Hull Analysis, the naive interpreter is modified so each head is associated with a set of constraints that are asserted along with the head. Proving the body of a clause is performed by substituting each operator with an equivalent constraint. If the set of constraints is satisfiable, it are projected onto the head, and the head is asserted with the constraints. In each iteration the constraints are *widened* to ensure termination of the analysis.

6.2 Query-Answer transformation

The specialised emulator is a Prolog program where data flow is propagated in a top-down fashion, and similarly, the execution strategy of the program is in a top-down manner. Furthermore, many PIC programs are not intended to terminate and "succeed" - they simply run forever. Such programs may have an empty model and so bottom-up analysis returns no useful information. Query-answer transformation provides a way to use a bottom-up analysis tool to return information about the computations themselves, in particular, on the set of calls to each predicate in the program [11].

We illustrate the query-answer transformation for the specialised emulator clauses. Take the clause shown in Example 3.

```
execute__5(B,A) :-
  C is B+A, 0 is C>>8,
  C \== 0, D is 24 /\ 254,
  execute__6(D,B,C).
```

If we are interested in obtaining information about the calls to `execute__6(D,B,C)` we write the following clause, which "inverts" the original clause.

```
execute__6_query(D,B,C) :-
  execute__5_query(B,A),
  C is B+A, 0 is C>>8,
  C \== 0, D is 24 /\ 254.
```

When provided with some initially called goal as a fact, typically `execute__0_query`, the bottom-up analyser generates a model for each query predicate.

6.3 Linear approximation

Each `execute`-clause of the specialised emulator consists of a number of equalities, inequalites, arithmetic and boolean operations (see Example 2). For the polyhedra operations the constraints must be linear expressions, so each of these operations must be approximated by a linear expression before they can be projected onto the clause head.

Equalities and inequalites: These are straightforward. The terms `X is Y` and `X = Y` both translates into the constraints $X = Y$. Similarly for the inequalities, e.g. `X >= Y` translates into the constraints $X \geq Y$. Terms containing not-equals are ignored since they have no linear approximation. The right hand side of the term contains the operations that needs approximation.

6.3.1 Approximation of arithmetic operations

For the arithmetic operations this is also straight forward. The addition and subtraction operations are both linear and e.g. `X is Y + Z` translates to $X = Y + Z$ *iff* $linear(Y) \wedge linear(Z)$.

The multiplication operator can also be approximated by a linear constraint, if either of the operands is a constant. So the term `X is Y * Z` translates into $X = Y \times Z$ *iff* $const(Y) \vee const(Z)$. At present, expressions that are not linear are simply ignored, so that their variables are completely unconstrained. Of course it is possible to make linear approximations in many cases but we have not yet applied these techniques.

6.3.2 Approximation of boolean operations

In this category are the boolean operators `AND`, `OR`, `NOT`, left and right bit shift.

The approximation of the unary operator `NOT` depends on the maximum integer size of the abstract machine we

are modelling. In our case, the 8 bit PIC processor, the max. integer value is 255.

The term `X is \Y` (bitwise negation) is approximated by the expression $X = 255 - Y$.

For the `AND` operator it is not possible to give an exact numerical approximation if either of the operands is not known. The results however can never be greater than smaller of the two operands, and never less than zero. The term `X is Y /\ Z` can be approximated by the constraints $X \leq Y \wedge X \leq Z \wedge X \geq 0$.

Similarly with the `OR` operator. If either of the operands is not known, an exact approximation cannot be given. The result of an `OR` operation however, can never be greater than the sum of the operands and never smaller than the largest operand. The term `X is Y \/ Z` can be approximated by the constraints $X \leq Y + Z \wedge X \geq Y \wedge X \geq Z$.

These constraints are not very precise. As shown in Example 5 the constraints on registers whose values are the result of a series of boolean operations, are correct but not precise.

Example 5 *This example is a small PIC program containing a loop of 10 iterations in which a register in each iteration is incremented by two. This is implemented with an* `OR` *and an* `ADD` *instruction. The instruction* `decfsz` *decrements* `CNT` *and skips the following instruction if the result is* 0.

```
1:    movlw D'10'
2:    movwf CNT
3:    movlw D'0'
4:    movwf SUM
5:    movlw D'1'
6:  loop1
7:    iorwf SUM,1
8:    addwf SUM,1
9:    decfsz CNT
10:    goto loop1
11:    goto MAIN
```

Running it through the analyser produces the constraints shown below; only the constraints on the instructions in the loop are interesting in this case. Multiple (possibly disjoint) sets of constraints can be generated for each instruction. The constraints are on the registers prior to the execution of the listed instruction. `Acc` *is the accumulator - its value is* 1 *from instruction 5 and to the end of the program. The actual value of* `SUM` *after the loop terminates, is 20. It is less than or equal to* 20 *as the constraints show, but it is not precise.*

```
7:    CNT=10,SUM=0
7:    Acc*CNT <= 9,2*CNT+ Acc*SUM <= 20
8:    Acc*SUM=1,CNT=10
8:    Acc*CNT <= 9,2*CNT+ Acc*SUM <= 21
9:    Acc*SUM=2,CNT=10
9:    Acc*CNT <= 9,2*CNT+ Acc*SUM <= 22
10:    Acc*CNT <= 9,2*CNT+ Acc*SUM <= 20
11:    Acc*SUM<= 20
```

Bit shift: The PIC processor has a bit shift instruction for left shifting bits and and operation for right shifting bits. Bits can only be shifted one position at a time. Left-shifting bits one position equals multiplication by 2 - this case has already covered under the arithmetic operations. Right-shifting is division by 2 and rounding down.

6.4 Widening

In the numerical domain arithmetic operations have no upper or lower bounds. For infinite chains of operations some mechanism must be implemented to ensure or accelerate the convergence of the fixpoint computations. This is what the polyhedral widening operator ensures. Various widening operators are provided by the PPL library. Applying these operators generally means losing precision.

In our specialised emulator loops can occur - even non terminating loops. Widening will ensure termination of the convex hull analysis, but at the cost of precision. Typically this will be evident in a lack of either upper or lower bounds.

Example 6 *Take for example the small PIC program shown below; a simple loop of 10 iterations, in which 10 is added to a register in each iteration of the loop (when the loop terminates the register contains the value 100).*

```
1:    movlw D'0'
2:    movwf SUM
3:    movlw D'10'
4:    movwf CNT
5:  loop1
6:    addwf SUM,1
7:    decfsz CNT
8:    goto loop1
9:    goto MAIN
```

The register containing the loop counter (`CNT`*) lacks the lower bound "1" inside the loop. Apart from that, the register containing the sum of 10's, has the right relation with the loop counter. Isolating* `SUM` *in the constraints for instruction 7 yields the follwing equation:* $SUM = 110 - 10 * CNT$*, for* $CNT \in [1...10] \Rightarrow SUM \in [10, 20, 30...100]$ *or the exact values SUM will be assigned in the loop.*

```
6:  1*Acc=10,CNT=10,SUM=0
6:  1*CNT<= 9,10*CNT+1*SUM=100,1*Acc=10
7:  1*SUM=CNT,1*Acc=10,CNT=10
7:  1*CNT<= 9,10*CNT+1*SUM=110,1*Acc=10
8:  1*CNT<= 9,10*CNT+1*SUM=100,1*Acc=10
9:  1*SUM=100,1*Acc=10
```

The widening problem is a general problem with convex hull analysis. Techniques for enhancing precision of widening, such as delayed widening, could also be used in our analyser. This has not yet been implemented.

7 Results

The entire procedure of doing a convex hull analysis of a PIC program involves several steps. This section gives an overview of how costly each of these steps are. Also included are results from the liveness analysis and the the dead code detection. Three PIC programs supplied by University of Bristol are used to evaluate the procedure.

The results from the liveness and dead code analysis are shown in Table 1. The column showing *Used Data Registers* shown how many data register that actually - at some point during execution of the program - will contain live data. The column *Remapped Registers* shown how many registers are actually needed, if the registers are allocated using a graph coloring technique [5].

Table 2 shows the execution time for each step in the analysis process - the Control Flow Analysis, the Partial Evaluation step, the Query-Answer transformation, the reverse Redundant Argument Filtering, the register remapping using graph coloring and the Convex Hull analysis. All analysis tools are implemented in Ciao Prolog with the exception of the Control Flow Analysis for which we use the *bddbddb*-package. The Parma Polyhedra Library used in the Convex Hull analysis step is implemented in C++, but it comes with a Prolog interface. All test were executed on the same machine, a Pentium III 1GHz with 256MB RAM running Linux. Timing results were collected using the command line tool *time*, counting the user CPU time.

8 Related work

The Hoist [20] project is closely related to our work. This project is also based on applying abstract interpretation to embedded software, to aid the programmer in producing reliable and efficient programs. In this project two domain are explored; an interval domain and a bitwise domain. Hoist does not rely on the programmer writing a simulator for the target processor, but can use existing simulators to automatically construct abstract operations. Hoist does not capture numerical relations between data registers and is limited to 8 bit machines.

9 Future Work

The emulator can be extended or "instrumented" with additional state information. For example a global clock could be added to the state. Applying the convex hull analyser to this extended emulator, could be used to approximate Worst Case Execution Time (WCET), where execution time would depend on values in registers e.g. loop counters or input values.

In this paper we focused on a numerical abstraction. This domain is not well suited for programs relying heavily on boolean operations. Other abstraction can be applied to the specialised emulator to give more precise results. A different domain being explored is a bit-size domain based on regular types and pre-interpretations [12, 13]. Registers are assigned a type based on which bit is the most significant bit in the value contained in the registers. The result of a boolean operation, take `OR` as an example, between to registers, is the largest bit-size of the two operands. Similarly bit shift increases or reduces the bit-size. Overflows would be detected by bit-sizes larger than the machine's bit size. This abstraction can complement the numerical abstraction in order to produce more precise results. Upper and lower bounds of boolean operations can be added to the constraints based on the bit-sizes obtained from the pre-interpretation.

10 Conclusion

We have shown how a set of general-purpose logic program transformation tools can be applied effectively to the analysis of programs for a simple microprocessor. While some of the analyses are standard, the advantage of this approach lies in the relative ease of developing tools for a new language and processor, and taking advantage of highly-developed libraries attached to logic program analysis toolsets, such as the PPL and the Datalog model evaluation tools. We also claim that the approach allows new analyses for logic programs, which are emerging constantly, to be immediately applied to other target languages. Furthermore, techniques for combining existing logic program analyses using the abstract interpretation framework can gain precision compared to the individual analyses (the reduced product approach [8]). Again, such techniques can be applied directly to the logic representations of other languages. We intend to combine boolean approximation domains with numeric approximation in the PIC case study.

Note that the PIC (or other target language) programmer need not be aware of the existence of the logic programming layer implementing the analysis. In our prototype implementation, a front- and back-end have been implemented so that the translation to Prolog and the transfer of the analysis results back to the PIC program are done transparently.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] R. Bagnara, E. Ricci, E. Zaffanella, and P. M. Hill. Possibly not closed convex polyhedra and the parma polyhedra library. In *SAS '02: Proceedings of the 9th International Symposium on Static Analysis*, pages 213–229, London, UK, 2002. Springer-Verlag.

Program	No. Instr.	Dead Instr.	Used Data Regs.	Remapped Regs.
comp_icp	141	0	3	2
acc_icsp	215	34	5	3
gps_smt	400	2	23	20

Table 1. Liveness and dead code analysis results.

Program	No. Instr.	CFA	PE	FAR	QA	Reg. Remap	Convex Hull	**Total Time**
comp_icp	141	3.5	13	2.3	1.3	0.4	5.3	**25.8**
acc_icsp	215	4.5	18	1.5	2.9	0.5	7.1	**24.5**
gps_smt	400	9.0	42	18.6	5.7	0.5	20.0	**95.8**

Table 2. Convex Hull analysis. Timing results are listed in seconds.

[3] F. Benoy and A. King. Inferring argument size relationships with CLP(r). In *Logic Program Synthesis and Transformation*, pages 204–223, 1996.

[4] F. Benoy, A. King, and F. Mesnard. Computing convex hulls with a linear solver.

[5] P. Briggs. Register allocation via graph coloring. Technical Report TR92-183, 24, 1998.

[6] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla. The Ciao prolog system. reference manual. Technical Report CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM), August 1997. Available from http://www.clip.dia.fi.upm.es/.

[7] M. Codish. Efficient goal directed bottom-up evaluation of logic programs. *The Journal of Logic Programming*, 38(3):354–370, 1999.

[8] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the 6th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio,Texas*, pages 269–282. ACM Press, New York, U.S.A., 1979.

[9] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th Annual ACM Symposium on Principles of Programming Languages*, pages 84–96, 1978.

[10] D. De Schreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens, and M. H. Sørensen. Conjunctive Partial Deduction: Foundations, control, algorithms, and experiments. *Journal of Logic Programming*, 41:231–277, November 1999.

[11] S. K. Debray and R. Ramakrishnan. Abstract interpretation of logic programs using magic transformations. *J. Log. Program.*, 18(2):149–176, 1994.

[12] J. P. Gallagher and K. S. Henriksen. Abstract domains based on regular types. In V. Lifschitz and B. Demoen, editors, *Proceedings of the International Conference on Logic Programming (ICLP'2004)*, volume 3132 of *Springer-Verlag Lecture Notes in Computer Science*, pages 27–42, 2004.

[13] J. P. Gallagher, K. S. Henriksen, and G. Banda. Techniques for scaling up analyses based on pre-interpretations. In M. Gabbrielli and G. Gupta, editors, *Proceedings of the 21st International Conference on Logic Programming, ICLP'2005*, volume 3668 of *Springer-Verlag Lecture Notes in Computer Science*, pages 280–296, 2005.

[14] K. S. Henriksen and J. P. Gallagher. Analysis and specialisation of a pic processor. In *Proceedings of the 2004 IEEE Conference on Systems, Man and Cybe rnetics*, The Hague, Netherlands, October 10-13 2004.

[15] M. Hermenegildo, F. Bueno, G. Puebla, and P. López. Program analysis, debugging, and optimization using the ciao system preprocessor.

[16] A. King, K. Shen, and F. Benoy. Lower-bound time-complexity analysis of logic programs. In *ILPS '97: Proceedings of the 1997 international symposium on Logic programming*, pages 261–275, Cambridge, MA, USA, 1997. MIT Press.

[17] M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In C. Li, editor, *PODS*, pages 1–12. ACM, 2005.

[18] M. Leuschel and J. Jørgensen. Efficient specialisation in Prolog using the hand-written compiler generator LOGEN. *Elec. Notes Theor. Comp. Sci.*, 30(2), 1999.

[19] M. Leuschel and M. H. Sorensen. Redundant argument filtering of logic programs. In *Logic Program Synthesis and Transformation*, pages 83–103, 1996.

[20] J. Regehr and A. Reid. Hoist: a system for automatically deriving static analyzers for embedded systems. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 133–143, New York, NY, USA, 2004. ACM Press.

[21] J. Ullman. *Principles of Knowledge and Database Systems; Volume 1*. Computer Science Press, 1988.

[22] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In W. Pugh and C. Chambers, editors, *PLDI*, pages 131–144. ACM, 2004.

[23] J. Whaley, C. Unkel, and M. S. Lam. A bdd-based deductive database for program analysis, 2004. `http://bddbddb.sourceforge.net/`.