Analysis of Imperative Programs through Analysis of Constraint Logic Programs

Julio C. Peralta, John P. Gallagher, and Hüseyin Sağlam

University of Bristol
Dept. of Computer Science
Merchant Venturers Building
Woodland Rd., Bristol, U.K. BS8 1UB
fax: +44-117-9545208
e-mail: {jperalta, john}@cs.bris.ac.uk, saglam@osf02.ktu.edu.tr

Abstract. In this paper a method is proposed for carrying out analysis of imperative programs. We achieve this by writing down the language semantics as a declarative program (a constraint logic program, in the approach shown here). We propose an effective style of writing operational semantics suitable for analysis which we call *one-state small-step* semantics. Through controlled partial evaluation we are able to generate residual programs where the relationship between imperative statements and predicates is straightforward. Then we use a static analyser for constraint logic programs on the residual program. The analysis results are interpreted through program points associating predicates in the partially evaluated interpreter to statements in its corresponding imperative program. We used an analyser that allows us to determine linear equality, inequality and disequality relations among the variables of a program without user-provided inductive assertions or human interaction. The proposed method intends to serve as a framework for the analysis of programs in any imperative language. The tools required are a partial evaluator and a static analyser for the declarative language.

Keywords: Partial Evaluation, Constraint Logic Programming, Operational Semantics, Imperative Program Analysis.

1 Introduction

Program semantics has long been used as a formal basis for program manipulation. By this we mean that the formal semantics of a programming language is written down in some mathematical framework, which is then used to establish program properties such as termination, correctness with respect to specifications, or the correctness of program transformations.

In the case of imperative languages the gap between semantics and programs is greater than in the case of declarative languages. Perhaps for this reason, semantics-based tools for declarative languages, such as abstract interpreters and partial evaluators, have been the subject of more intense research than similar tools for imperative languages.

G. Levi (Ed.): SAS'98, LNCS 1503, pp. 246–261, 1998.

The aim of this paper is to show how to transfer results achieved in declarative languages to imperative languages. The approach is to implement the semantics of imperative languages carefully in a declarative language for which analysis and transformation tools exist.

There exist several kinds of semantics for imperative languages. The choice of which one is better suited for the particular application is a subject of ongoing research [3]. We shall see that for the purposes of this paper a variant of operational semantics will suffice.

Logic programming appears to be an elegant paradigm for imperative language semantics implementation. When written carefully, the semantics can be regarded as an *interpreter* for the imperative language. Appropriate techniques for implementing semantics are discussed in Sect. 2. Partial evaluation of the interpreter with respect to an imperative program yields an equivalent declarative program. By so doing we open up the possibility of applying well-developed techniques for analysis and transformation of constraint logic programs to imperative programs as well. Nevertheless, it is not clear how to relate the results of such analysis and/or transformation back to the original imperative program.

In order to obtain a correspondence between the imperative program and its corresponding declarative program some tuning of the partial evaluator is needed. Otherwise, the partial evaluator may remove important information needed to relate imperative statements and variables with their declarative counterpart. Such tuning involves selecting among the predicates of the semantics-based interpreter those we want to be defined in the residual program. Hence, we choose predicates from the semantics-based interpreter that relate directly to the meaning of the statements in the imperative program to be partially evaluated. As a result we get one predicate for each statement of the imperative program, thus highlighting the correspondence between imperative statements and predicates in the residual program.

In this paper we propose a method that intends to serve as a framework for the analysis of programs in any imperative language, by writing down its semantics as a declarative program (a constraint logic program, in the approach shown here). The tools required are a partial evaluator and a static analyser for the declarative language.

Section 2 considers the overall problem of encoding semantics in a logic programming language, in a form suitable for analysis. Section 3 provides some remarks on the implementation of the operational semantics for a small imperative language [1]. In Sect. 4 we show the partial evaluation process and give an example. Section 5 illustrates how to relate the results of the analysis back to its imperative program in a systematic way. Finally, in Sects. 6 and 7 we discuss related work, and state our final remarks and some trends for future work.

[1] Currently we are writing the semantics for Java in a similar style

2 Analysis through Semantics

Before describing our experiments in detail let us consider some critical points concerning representing semantics in a form suitable for analysis. There are several styles of semantics for imperative languages to be found in textbooks. These may all be translated more or less directly into declarative programming languages, but it is necessary to consider carefully the choice of semantics and the style in which the semantics is represented. The choice is influenced by two questions: firstly, what kind of analysis is to be performed on the imperative program, and secondly, how can the complexity of the analysis be minimised?

2.1 Big-Step and Small-Step Semantics

The usual distinction between different kinds of semantics is between the compositional and operational styles. However for our purpose, the most relevant division is between *big-step* and *small-step* semantics. Note that operational semantics can be either big-step (natural semantics) or small-step (structural operational semantics).

Let us represent program statements by S and computation states by E. Big-step semantics is modelled using a relation of the form $bigstep(S, E_1, E_2)$, which means that the statement S transforms the initial state E_1 to final state E_2. The effect of each program construct is defined independently; compound statements are modelled by composing together the effects of its parts.

On the other hand, small-step semantics is typically modelled by a transition relation of the form $smallstep(\langle S_1, E_1\rangle, \langle S_2, E_2\rangle)$. This states that execution of statement S_1 in state E_1 is followed by the execution of statement S_2 in state E_2. Small-step semantics models a computation as a sequence of computation states, and the effect of a program construct is defined as its effect on the computation state in the context of the program in which it occurs.

Big steps can be derived from small steps by defining a special terminating statement, say *halt*, and expressing big-step relations as a sequence of small steps.

$$bigstep(S, E_1, E_2) \leftarrow smallstep(\langle S, E_1\rangle, \langle halt, E_2\rangle).$$
$$bigstep(S, E_1, E_3) \leftarrow smallstep(\langle S, E_1\rangle, \langle S_1, E_2\rangle), bigstep(S_1, E_2, E_3).$$

For the purposes of program analysis, the two styles of semantics have significant differences. Analysis of the *bigstep* relation allows direct comparison of the initial and final states of a computation. As shown above, big steps are derivable from small steps but the analysis becomes more complex. If the purpose of analysis is to derive relationships between initial and final states, then big-step semantics would be recommended.

On the other hand, the *smallstep* relation represents directly the relation between one computation state and the next, information which would be awkward (though not impossible) to extract from the big-step semantics. Small-step semantics is more appropriate for analyses where local information about states is required, such as the relationship of variables within the same state, or between successive states.

2.2 One-State and Two-State Semantics

There is another option for representing small-step semantics, which leads to programs significantly simpler to analyse. Replace the $smallstep(\langle S_1, E_1\rangle, \langle S_2, E_2\rangle)$ relation by a clause $exec(S_1, E_1) \leftarrow exec(S_2, E_2)$. The meaning of the predicate $exec(S, E)$ is that there exists a terminating computation of the statement S starting in the state E. We also have to add a special terminal statement called *halt* which is placed at the end of every program, and a statement $exec(halt, E) \leftarrow true$.

We call this style of small-step semantics a *one-state* semantics since the relation *exec* represents only one state, in contrast to *two-state* semantics given by the *bigstep* and *smallstep* relations.

As an aside, the one-state and two-state styles of representation follow a pattern identified by Kowalski in [10], when discussing graph-searching algorithms in logic programming. Kowalski noted that there were two ways to formalise the task of searching for a path from node a to node z in a directed graph. One way is to represent the graph as a set of facts of the form $go(X, Y)$ representing arcs from node X to node Y. A relation $path(X, Y)$ could then be recursively defined on the arcs. The search task is to solve the goal $\leftarrow path(a, z)$. Alternatively, an arc could be represented as a clause of form $go(Y) \leftarrow go(X)$. In this case, to perform the task of searching for a path from node a to node z, the fact $go(a) \leftarrow true$ is added to the program, and computation is performed by solving the goal $\leftarrow go(z)$. There is no need for a recursively defined path relation, since the underlying logic of the implication relation fulfils the need.

One-state semantics corresponds to the use of the relation $go(X)$ while two-state semantics corresponds to using the relation $go(X, Y)$. The "start state" corresponds to the clause $exec(halt, E) \leftarrow true$. Our experiments show that the one-state semantics is considerably simpler to analyse than the two-state semantics.

2.3 Analysis of the Semantics

It may be asked whether one-state semantics is expressive enough, since no output state can be computed. That is, given a program P and initial state E, the computation is simulated by running the goal $\leftarrow exec(P, E)$ and the final state is not observable. This is certainly inadequate if we are using our semantics to simulate the full effect of computations.

However, during program analysis of $\leftarrow exec(P, E)$ more things are observable than during normal execution of the same goal. In particular we can use a program analysis algorithm that gives information about calls to different program subgoals. In one-state semantics with a relation $exec(S, E)$, the analysis can determine (an approximation of) every instance of $exec(S, E)$ that arises in the computation. We can even derive information about the relation of successive states since the analysis can derive information about instances of a clause $exec(S_1, E_1) \leftarrow exec(S_2, E_2)$.

In the experiments to be described below, we start from a structural operational semantics in a textbook style, and derive a one-state small-step semantics.

3 A Semantics-Based Interpreter

As already mentioned a desirable property of structural operational semantics is the way it reflects every change in the computation state. Here we present briefly a way of systematically translating formal operational semantics (adapted from [16]) into a constraint logic program.

We shall first provide some elements of the syntax of the imperative language and the metavariables used in the semantics descriptions. Assume we have an imperative language L with assignments, arithmetic expressions, while statements, if-then-else conditionals, empty statement, statement composition[2], and boolean expressions. Let S be a statement, a be an arithmetic expression, b a boolean expression, e a variable environment (mapping variables to their value), and x a program variable. All these variables may occur subscripted. A pair $\langle S, e\rangle$ is a *configuration*. Also a state is a special terminal configuration. The operational semantics below give meaning to programs by defining an appropriate transition relation holding between configurations.

3.1 Structural Operational Semantics

Using structural operational semantics [16] a transition relation $\Rightarrow$ defines the relationship between successive configurations. There are different kinds of transition corresponding to different kinds of statement. Accordingly, the meaning of empty statement, assignment statement, if-then-else statement, while-do statement and composition of statements are:

$$\langle \mathit{skip}, e\rangle \Rightarrow e \tag{1}$$

$$\langle x := a, e\rangle \Rightarrow e[x \mapsto \mathcal{A}[\![a]\!]e] \tag{2}$$

$$\langle \mathit{if}\ b\ \mathit{then}\ S_1\ \mathit{else}\ S_2, e\rangle \Rightarrow \langle S_1, e\rangle \text{ if } \mathcal{B}[\![b]\!]e = \mathbf{tt} \tag{3}$$

$$\langle \mathit{if}\ b\ \mathit{then}\ S_1\ \mathit{else}\ S_2, e\rangle \Rightarrow \langle S_2, e\rangle \text{ if } \mathcal{B}[\![b]\!]e = \mathbf{ff} \tag{4}$$

$$\langle \mathit{while}\ b\ \mathit{do}\ S, e\rangle \Rightarrow \langle \mathit{if}\ b\ \mathit{then}\ (S; \mathit{while}\ b\ \mathit{do}\ S)\ \mathit{else}\ \mathit{skip}, e\rangle \tag{5}$$

$$\langle S_1; S_2, e\rangle \Rightarrow \langle S_1'; S_2, e'\rangle \text{ if } \langle S_1, e\rangle \Rightarrow \langle S_1', e'\rangle \tag{6}$$

$$\langle S_1; S_2, e\rangle \Rightarrow \langle S_2, e'\rangle \text{ if } \langle S_1, e\rangle \Rightarrow e' \tag{7}$$

where $\mathcal{A}[\![\cdot]\!]$ is the semantics function for arithmetic expressions and $\mathcal{B}[\![\cdot]\!]$ is the semantics function for boolean expressions. Intuitively, the assignment axiom schema above says that in a state e, $x := a$ is executed to yield a state $e[x \mapsto \mathcal{A}[\![a]\!]e]$ which is as e except that x has the value $\mathcal{A}[\![a]\!]e$. Moreover, transition 6 expresses that if S_1 is not a primitive statement of the language then execution won't proceed to S_2 until the rest of S_1, S_1', has been fully executed. Transition 7 considers the case when execution of S_1 has been completed thus yielding state e', hence execution of S_2 starts from this new state.

We may specialise transitions 6 and 7 by unfolding their conditions with respect to transitions 1, 2, 3, 4, and 5 above to transitions 9, 10, 11, 12, and 13

[2] The statement composition operator ';' is assumed to be right associative

below. Transition 8 below is obtained by unfolding the condition of transition 6 with respect to itself and applying the associativity property of the ';' operator. We assume that cases 2, 3, 4 and 5 do not occur since all programs are terminated by *skip*. Hence the new semantics:

$$\langle \mathit{skip}, e\rangle \Rightarrow e$$

$$\langle (S_1; S_2); S_3, e\rangle \Rightarrow \langle S_1; (S_2; S_3), e\rangle \quad (8)$$

$$\langle \mathit{skip}; S_2, e\rangle \Rightarrow \langle S_2, e\rangle \quad (9)$$

$$\langle x := a; S_2, e\rangle \Rightarrow \langle S_2, e[x \mapsto \mathcal{A}[\![a]\!]e]\rangle \quad (10)$$

$$\langle (\mathit{if}\ b\ \mathit{then}\ S_1\ \mathit{else}\ S_2); S_3, e\rangle \Rightarrow \langle S_1; S_3, e\rangle \text{ if } \langle \mathcal{B}[\![b]\!]e = \mathbf{tt}\rangle \quad (11)$$

$$\langle (\mathit{if}\ b\ \mathit{then}\ S_1\ \mathit{else}\ S_2); S_3, e\rangle \Rightarrow \langle S_2; S_3, e\rangle \text{ if } \langle \mathcal{B}[\![b]\!]e = \mathbf{ff}\rangle \quad (12)$$

$$\langle (\mathit{while}\ b\ \mathit{do}\ S); S_2, e\rangle \Rightarrow \quad (13)$$
$$\langle \mathit{if}\ b\ \mathit{then}\ (S; \mathit{while}\ b\ \mathit{do}\ S\ \mathit{else}\ \mathit{skip}); S_2, e\rangle$$

Next, we express the semantics as a constraint logic program below. It is worth noting that this representation aids the analysis phase by carrying a single environment instead of double environment (that is, the one-state small-step semantics discussed in Sect. 2).

```
exec(skip,E) <-
exec(compose(compose(S1,S2),S3),E) <-
      exec(compose(S1,compose(S2,S3)),E)
exec(compose(skip,S2),E) <- exec(S2,E)
exec(compose(assign(X,Ae),S2),E) <- a_expr(E,Ae,V),
                                    update(E,E2,X,V),
                                    exec(S2,E2)
exec(compose(ifte(B,S1,S2),S3),E) <- b_expr(E,B,true),
                                     exec(compose(S1,S3),E)
exec(compose(ifte(B,S1,S2),S3),E) <- b_expr(E,B,false),
                                     exec(compose(S2,S3),E)
exec(compose(while(B,S1),S2),E) <-
      exec(compose(ifte(B,compose(S1,while(B,S1)),skip),skip),E)
```

In this program the term `assign(X,A)` represents an assignment statement `X := A`, and the term `compose(S1,S2)` represents the composition of statement `S1` with statement `S2`. The predicate `exec(P,E)` holds iff program `P` can be executed from state `E`. The predicate `update(E1,E2,X,V)` models the change of state from `E1` to `E2` induced by the assignment of `V` to `X`. The value of the assigned variable `X` is changed to `V` but all other values of variables are passed unchanged from `E1` to `E2`. Observe that the clause for composition of statements in this implementation has been specialised to the individual cases of our program constructs, namely `assign, skip, ifte` and `while` in this example. Therefore we do not use the clauses corresponding to composition of statements in their general form.

One way to enhance the partial evaluation and the analysis of the semantics-based interpreter is to add constraints in the definition of predicates that handle arithmetic. For instance, define `a_expr` for addition of arithmetic expressions as:

```
a_expr(E,plus(Ae1,Ae2),V) <- a_expr(E,Ae1,V1), a_expr(E,Ae2,V2),
                             {V=V1+V2}
```

where the curly brackets are used to invoke the constraint solver. This enables constraint-based techniques to be applied [18], [20].

4 Partial Evaluation

Partial evaluation generates a *residual program* by *evaluating* a program with *part* of its input which is available at compile-time. When given the remaining part of the input, the residual program produces the same output as the source program when presented with the whole input. The aim of partial evaluation is to gain efficiency by specialising the source program, exploiting the known input.

A partial evaluator is applied to our semantics-based interpreter with respect to an input imperative program. The result is a residual logic program which is a version of the semantics interpreter specialised with respect to the input program. There is a well known relationship between partial evaluation and compilation, and the residual program can be viewed as the result of compiling the imperative program into the declarative language in which the semantics interpreter is written (a constraint logic program in our case). The residual program is then analysed, and the analysis results are related to the original imperative program, as we will discuss.

The role of partial evaluation is two-fold. One is to increase the efficiency and precision of the analysis. The other is to allow the results of the analysis to be mapped directly to the structure of the original imperative program. Analysis of imperative programs commonly associates a set of assertions with program points occurring between statements. By contrast, our top-down analyser for constraint logic programs produces a separate analysis per program predicate. This suggests having a predicate in the residual program corresponding to each program point in the imperative program.

Thus, for the purposes of analysis, partial evaluation is not required to produce the maximum possible specialisation. It suffices to have a translator from imperative to logic programs where imperative statements and variables can be explicitly related to predicates and their arguments. Some special purpose control over standard partial evaluation is required, which will be described below, but first we need to describe the data structure which is used to represent the imperative program.

4.1 Representation of Imperative Programs

Starting from the standard textual representation of imperative programs the first step is to produce a list of tokens. As the next step in producing the desired

specialisation we add a unique label to each statement of the input imperative program. These labels are used in the control of the partial evaluation. The term representation is produced using a conventional LALR parser [1]. The parser takes for input the tokenised representation of the imperative program and produces as output a list of terms [3] suitable for interpretation by the semantics-based interpreter. Such a list of terms contain labels that will allow us to identify every statement in the imperative program. For instance, consider the following program fragment representing an if-then-else statement with one assignment in each conditional branch:

```
[if,a,>,b,then,
   x,:=,a,+,2,;,
 else,
   y,:=,b,-,1]
```

By parsing the above code we obtain:

```
[ifte(gt(a,b),
  [assign(x,plus(a,2))],
  [assign(y,minus(b,1))]
 )]
```

When decorated with program points we get:

```
[p(1,ifte(gt(a,b),
     p(2,[assign(x,plus(a,2))]),
     p(3,[assign(y,minus(b,1))])
 ))]
```

We use the function symbol `p` to enclose a program point. The first argument of `p` is a program point label and the second its associated program statement.

The semantics-based interpreter is modified to handle the newly extended representation of statements. For example, instead of

```
exec(compose(assign(X,Ae),S2),E1) <- a_expr(E1,Ae,V),
                                     update(E1,E2,X,V),
                                     exec(S2,E2)
```

we write

```
exec(compose(p(La,assign(X,Ae)),S2),E1) <- a_expr(E1,Ae,V),
                                           update(E1,E2,X,V),
                                           exec(S2,E2)
```

A similar modification is made for every semantics clause defining a program statement and its program point.

[3] For simplicity we use the list function symbol and Prolog list notation instead of the `compose` function symbol to denote composition of statements. Instead of `compose(S1,S2)` we write `[S1|S2]`.

4.2 Control of Partial Evaluation

Standard partial evaluation of logic programs [12], given a query and a program, seeks *a set of atoms* for which two properties hold, closedness and independence. To achieve both goals (closedness and independence) partial evaluators usually distinguish two levels of control:

- the *global control*, in which one decides which atoms will be partially evaluated, and
- the *local control*, in which one constructs the finite partial computation trees for each individual atom in the set and thus determines what the definitions of the partially evaluated atoms look like.

The global control determines whether the set of atoms contains more than one version of each predicate (*polyvariant control*) or whether only one version of every predicate is kept (*monovariant control*). We require polyvariant control since each program point should result in a distinct predicate in the residual program; for instance, a version of the semantics transition dealing with assignment statements should be produced for each assignment in the input program.

During partial evaluation global and local control interact, passing information between each other. For our purposes, we require the local control to evaluate the parts of the interpreter that break down the imperative program into its constituent statements (labelled program points).

The algorithm for partial evaluation that we use is based on the basic algorithm presented in [4], where we define the local control rule `Cr` and the abstraction operation `abstract` to cater for program points.

```
INPUT:  a program P and goal G, and local control rule Cr.
OUTPUT: a set of atoms

begin
   A[0] := the set of atoms in G
   i := 0
   repeat
      P' := a partial evaluation of A[i] in P using Cr
      R := A[i] U { p(t) | B <- Q,p(t),Q' in P'  OR
                           B <- Q,not(p(t)),Q' in P' }
      A[i+1] := abstract(R)
      i := i+1
   until A[i] = A[i-1] (modulo variable renaming)
end
```

The global control is provided by the operation `abstract(S)`. Our partial evaluator uses characteristic trees [6] as an aid in controlling the polyvariance and keeping the set of atoms finite at the global control level. The characteristic tree of an atom is a term capturing the shape of the computation tree produced for that atom using `Cr`.

Global control based on characteristic trees is modified to include the program point labels. The abstraction operation uses the following algorithm

- Let R be a finite set of atoms. Let R_{label} be the set of atoms in R which contain an argument of the form `p(N,Stm)`, where `Stm` is one of `assign`, `ifte` or `while`, and `N` is a program point label. Let R_{chtree} be the remaining atoms in R.
- Let $\{R_{N_1}, \ldots, R_{N_k}\}$ be the finite partition of R_{label} where all atoms in R_{N_j} contain the argument `p(Nj,Stm)`.
- Let $\{R_{c_1}, \ldots, R_{c_m}\}$ be the finite partition of R_{chtree} where R_{c_i} is the set of atoms in R_{chtree} having characteristic tree c_i.
- Let $abstract(R) = \{msg(R_{N_1}), \ldots, msg(R_{N_k}), msg(R_{c_1}), \ldots, msg(R_{c_m})\}$.

This is a correct abstraction according to the conditions in [4]. The purpose of the abstraction is to preserve a separate atom for each predicate that manipulates a program point, and to use characteristic tree abstraction for the other atoms. Note that the most specific generalisation (msg) preserves the program point argument.

At the local control level we use a determinate unfolding rule that generates SLDNF-trees with shower shape in the sense of [11, p. 36], suspending the evaluation whenever the semantics handles a program point. Some polyvariance and thus specialisation can be lost compared to an abstraction based on characteristic trees alone, but the result is appropriate for the needs of the analysis.

Example. Consider the contrived imperative program

```
(1)  i := 2;
(2)  j := 0;
(3)  while (n*n > 1) do
(4)    if (n*n = 2) then
(5)      i := i+4;
        else
(6)      i := i+2;
(7)      j := j+1;
      endwhile
```

By partial evaluation with respect to the above program and unknown initial environment we obtain:

```
program(X1,X2,X3) <-
      assign_1(X1,X2,X3)
assign_1(X1,X2,X3) <-
      assign_2(X2,X3)
assign_2(X1,X2) <-
      loop_1(2,0,X2)
loop_1(X1,X2,X3) <-
      X4 is X3*X3,
      gt_test_1(X4,X5),
      do_1(X1,X2,X3,X5)
gt_test_1(X1,tt) <- X1>1
```

```
cond_1(X1,X2,X3) <-
     X4 is X3*X3,
     eq_test_1(X4,X5),
     cnd_1(X1,X2,X3,X5)
cnd_1(X1,X2,X3,tt) <-
     assign_6(X1,X2,X3)
cnd_1(X1,X2,X3,ff) <-
     assign_7(X1,X2,X3)
assign_6(X1,X2,X3) <-
     X4 is X1+4,
     loop_1(X4,X2,X3)
```

```
gt_test_1(X1,ff) <- X1=<1
eq_test_1(X1,tt) <- X1=:=2
eq_test_1(X1,ff) <- X1=\=2
do_1(X1,X2,X3,ff) <-
do_1(X1,X2,X3,tt) <-
      cond_1(X1,X2,X3)
```

```
assign_7(X1,X2,X3) <-
     X4 is X1+2,
     assign_8(X4,X2,X3)
assign_8(X1,X2,X3) <-
     X4 is X2+1,
     loop_1(X1,X4,X3)
```

where the predicate name prefixes `assign`, `cond`, and `loop` denote assignment, if-then-else, and while statements respectively. Here the correspondence between predicates and program points is as follows; (this information can be extracted automatically from the partial evaluator).

Predicate	Program point	Predicate	Program point
`assign_1`	(1)	`assign_6`	(5)
`assign_2`	(2)	`assign_7`	(6)
`loop_1`	(3)	`assign_8`	(7)
`cond_1`	(4)		

5 Results

Decorating statements in the imperative program with unique identifiers leads to ways of relating the imperative program to a residual CLP program. That is, for every program point of the imperative program we generate a clause head in the corresponding residual program thus providing information of how the specialised CLP program relates to the imperative program. Then by using logic program analysis tools we can obtain information from the CLP residual program. Through the program points it is straightforward to translate the results of the CLP program analysis to an imperative program analysis.

Example (continued): Once we have an appropriate constraint logic program representing an imperative program with the desired clause heads we input this program to the analyser for constraint logic programs [18]. By analysing the residual program shown at the end of Sect. 4.2 we obtain the following results:

```
program_query(X1,X2,X3).
assign_1_query(X1,X2,X3).
assign_2_query(X1,X2).
assign_6_query(X1,X2,X3) :-X2>=0.0,X2-0.5*X1=< -1.0.
loop_1_query(X1,X2,X3) :-X2>=0.0,X2-0.5*X1=< -1.0.
cond_1_query(X1,X2,X3) :-X2>=0.0,X2-0.5*X1=< -1.0.
assign_7_query(X1,X2,X3) :-X2>=0.0,X2-0.5*X1=< -1.0.
assign_8_query(X1,X2,X3) :-X2>=0.0,X2-0.5*X1=< -2.0.
```

where we have a set of constraints associated with each predicate in the analysed program[4]. The suffix `_query` in every predicate name indicates that the

[4] We show the results only for the relevant predicates

constraints hold for the variables named every time we call that predicate during program execution. The query predicates are produced by query-answer transformation [7] applied to the partially evaluated program and the initial goal. This transformation takes a program and a goal and constructs definitions of the subgoals and their answers that arise during the computation of the goal.

Since we have a clause defining a predicate associated with a program point we may then say that the constraints associated with a predicate query are those that hold before the program point to which that predicate refers.

Once we have identified which is the relationship holding between the predicates and the program points it remains to determine how variables relate.

Partial evaluation performs some transformations regarding constant arguments and multiple occurrence of the same variable in an atom, when constructing the residual program. *Argument filtering and flattening* [5] are applied by our partial evaluator. For the purposes of our analysis we have suppressed them because they obscure the correspondence between logic programs and their imperative counterpart. For instance, the clause head for the predicate `assign_6` in the last example of Sect. 4.2 without redundant argument filtering and flattening is:

```
assign_6(5, [[i,j,n],[X1,X2,X3]], i, plus(i,4),
[p(3,while(gt(times(n,n),1),
   [p(4,ifte(eq(times(n,n),2),
       [p(5,assign(i,plus(i,4)))],
       [p(6,assign(i,plus(i,2))),
        p(7,assign(j,plus(j,1)))]
 ))]))]).
```

The application of both these transformations to the above atom yields the atom `assign_6(X1,X2,X3)`. We can see in the original atom above that the first argument is the imperative program point and the second is the variable environment relating imperative and logic variables for the scope of that predicate's clause. In our case we use both versions of the residual program. Namely, we use the small version for the analyser and the verbose one to systematically relate program points to clause heads and variables in both worlds. Given the variable environment we may readily obtain the correspondence between logic variables and imperative variables, and between clause heads and program points. Interpreting the above results yields:

```
(1)  i := 2;
(2)  j := 0;
                  {j>=0, 2j+2=<i}
(3)  while (n*n > 1) do
                  {j>=0, 2j+2=<i}
(4)    if (n*n = 2) then
                  {j>=0, 2j+2=<i}
(5)      i := i+4;
       else
```

```
                {j>=0, 2j+2=<i}
(6)      i := i+2;
                {j>=0, 2j+4=<i}
(7)      j := j+1;
      endwhile
```

Another Example. Next we present another example using lists to denote arrays. This code sorts an array of size `n` using the bubblesort algorithm of [9]. This example was adapted from one used by Cousot and Halbwachs [2]. The analysis results appear as comments inside curly brackets along the code.

```
(1)  b := n;
(2)  while (b>=1) do
                          {n>=b,b>=1}
(3)       j := 1;
(4)       t := 0;
(5)       while (j =< b-1)
                          {n>=b,t>=0,j>=t+1,b>=j+1}
(6)          if (k[j] > k[j+1]) then
                          {n>=b,t>=0,j>=t+1,b>=j+1}
(7)             tm := k[j];
(8)             k[j] := k[j+1];
(9)             k[j+1] := tm;
(10)            t := j;
                          {n>=b,j>=1,t>=0,b>=j+1,j>=t}
(11)         j := j+1
(12)      if (t == 0) then
                          {n>=b,t=0,j>=1,b>=j,b<j+1}
(13)         b := -1;
          else
                          {n>=b,t>=0,j>=t+1,b>=j,b<j+1}
(14)         b := t;
```

These results are the same as those obtained in [2].

6 Related Work

The first practical results on imperative languages for deriving linear equality or disequality relations among the variables of a program is due to Cousot and Halbwachs [2]. Their system was implemented in Pascal. The model execution used is based on flow-charts and an approximation method based on convex polyhedra. Incidentally the analyser used on the experiments here reported uses a similar approximation method integrated with other constraint solvers [20]. Later on in [3] the author poses the possibility of deriving different static analysers parameterised by the language semantics. In a similar way [8] show how to

obtain a static analyser for a non strict functional language. Such a static analyser is derived by successive refinements of the original language specification, natural operational semantics. The possible analyses obtained by the analyser derived with this method depend on the program property sought. This program property should be provided in advance. It appears that this technique has been applied to obtain some classical compiler analyses of programs in the sense of [1]. A good source of related work on implementation/derivation of static analysers from operational semantics for different programming languages is [8]. In [21] the authors describe a technique based on the style of abstract interpretation to statically estimate the ranges of variables throughout a program. Their implementation has been realised in the context of an optimising/parallelising compiler for C. Again, this is an example of using a variant of operational semantics to describe the abstract interpreters for static analysis of imperative programs.

In [13] the author lays out the theory of abstract interpretation using two-level semantics. Two-level semantics had been previously used in [15] to describe code generation. A summary of both can be found in [14]. Using denotational definitions the semantics of Pascal-like languages is given making explicit the distinction between compile time computations and run time computations, hence the two levels of the metalanguage. For program analysis, an appropriate interpretation of the run time metalanguage aids the analysis by giving a nonstandard semantics to run time constructs. By contrast in the present work, the semantic definitions are given in a standard way, and the translation is carried out by the partial evaluator where the distinction between compile time and run time computations is accounted for.

Another sort of problem reduction for analysis is provided in [17]. The authors convert the problem of identifying dependences among program elements to a problem of identifying may-aliases. The transformation output is a program in the same language as the input program where may-aliases in the transformed program yield information directly translatable to control flow dependences between statements in the original program. In a similar way the authors claim that control flow dependences in the transformed program have a direct reading as may-aliases in the corresponding program. Presumably the their method and ours could be combined to obtain other problem reduction results.

In [19] it is shown how to use logic programs to aid the analysis of imperative programs with pointers. The formalism is shown for the case of the pointer equality problem in Pascal. During analysis a set of assertions, represented as unary clauses, is updated according to the meaning of the program statement evaluated, the update operation designated and a set of consistency rules. The update operations resemble operations in deductive databases. The semantics of the imperative program is not explicitly represented as a logic program as in our approach, but in both approaches logic programs are used to express program properties.

7 Final Remarks

We have developed a language-independent method for analysing imperative programs. The method is based on encoding the semantics of an imperative language in a logic programming language for which there are advanced tools available for program analysis and transformation. This allows us to transfer the results of research on analysis of logic programs to the analysis of imperative programs.

The emphasis of our work is to find practical and efficient techniques for achieving this aim. A key aspect is to write the semantics in a way that is amenable to analysis. We identified the one-state small-step semantics as a suitable style. The problem of relating the results of analysis of a logic program to the source code of the original imperative program was also solved. A representation of the imperative program was constructed in which program points were represented by special terms in the logic program. The partial evaluation algorithm was modified to exploit these terms, and thus produce a residual logic program whose structure mirrored that of the imperative program. Thus results of analysis of the logic programs could be related directly to the imperative code.

The correctness of our results follow from correctness of the partial evaluator and correctness of the analyser. Both correctness proofs may be done independently of the imperative language to be analysed, which we claim is one of the contributions of the present work.

Future Work

We have performed some promising experiments on a simple language, as shown in this paper, but our aim is to analyse programs in a mainstream imperative language. Currently we are well advanced in writing the operational semantics for a significant subset of Java. We aim to enhance our current analysis tools by handling non-linear arithmetic constraints, and boolean constraints. Moreover, we aim to increase the flexibility of analysis by using pre-interpretations to express properties and abstract compilation to encode them as logic programs [18].

The use of the same method to perform other analyses, such as context-sensitive or alias analyses remains to be explored.

References

1. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1986.
2. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints amog variables of a program. In *Proceedings of the Conference Record of the 5th ACM Symposium on Principles of Programming Languages*, pages 84–97, Albuquerque, New Mexico, 1978.
3. Patrick Cousot. Abstract interpretation based static analysis parametrized by semantics. In *Proceedings of the Fourth International Symposium on Static Analysis, SAS'97*, pages 388–394, Paris, France, 1997. LNCS 1302, Springer Verlag.

4. J.P. Gallagher. Tutorial on specialisation of logic programs. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–98, Copenhagen, Denmark, 1993. ACM Press.
5. J. Gallagher and M. Bruynooghe. Some low-level source transformations for logic programs. In *Proceedings of Meta90 Workshop on Meta Programming in Logic*. Katholieke Universiteit Leuven, Belgium, 1990.
6. J. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 9(3&4):305–333, 1991.
7. J. Gallagher and D.A. de Waal. Fast and precise regular approximation of logic programs. In P. Van Hentenryck, editor, *Proceedings of the International Conference on Logic Programming (ICLP'94), Santa Margherita Ligure, Italy*. MIT Press, 1994.
8. Valérie Gouranton and Daniel Le Métayer. Formal development of static program analysers. In *Proceedings of the 8th Israeli Conference on Computer Systems and Software Engineering*, pages 101–110, Israel, 1997.
9. Donald E. Knuth. *The Art of Computer Programming*, volume 3 of *Sorting and Searching*. Addison-Wesley Publishing Company, 1973.
10. Robert Kowalski. *Logic for Problem Solving*. North Holland, 1979.
11. Michael Leuschel. *Advanced Techniques for Logic Program Specialisation*. PhD thesis, Katholieke Universiteit Leuven, Department of Computer Science, Leuven, Belgium, May 1997.
12. J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11(3&4):217–242, 1991.
13. Flemming Nielson. Two-level semantics and abstract interpretation. *Theoretical Computer Science*, (69):117–242, 1989.
14. Flemming Nielson. Semantics-directed program analysis: A toolmaker's perspective. In *Third International Symposium, SAS'96*. Springer Verlag, LNCS 1145, 1996.
15. Flemming Nielson and Hanne Riis Nielson. Two-level semantics and code generation. *Theoretical Computer Science*, (56):59–133, 1988.
16. Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications*. John Wiley and Sons, 1992.
17. John L. Ross and Mooly Sagiv. Building a bridge between pointer aliases and program dependences. In *European Symposium On Programming*, Lisbon, Portugal, 1998.
18. Hüseyin Sağlam. *A Toolkit for Static Analysis of Constraint Logic Programs*. PhD thesis, Bristol University, Department of Computer Science, Bristol, U.K., March 1998.
19. Mooly Sagiv, Nissim Francez, Michael Rodeh, and Reinhard Wilhelm. A logic-based approach to program flow analysis. 1998. Submitted to Acta Informatica.
20. H. Sağlam and J. Gallagher. Constrained regular approximation of logic programs. In N. Fuchs, editor, *Logic Program Synthesis and Transformation (LOPSTR'97)*. Springer-Verlag, Lecture Notes in Computer Science, 1998. in press.
21. Clark Verbrugge, Phong Co, and Laurie Hendren. Generalized constant propagation a study of C. *Lecture Notes in Computer Science, Compiler Construction*, (1060):74–90, 1996.