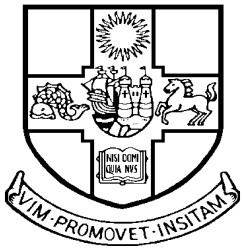
Towards Semantics-Based Partial Evaluation of Imperative Programs

Julio C. Peralta John P. Gallagher

April 1997 **CSTR-97-003**



University of Bristol

Department of Computer Science

Also issued as ACRC-97:CS-003

Towards Semantics-based Partial Evaluation of Imperative Programs

Julio C. Peralta and John P. Gallagher
University of Bristol, Dept. of Computer Science
e-mail: {jperalta,john}@cs.bris.ac.uk

Abstract

The semantics of an imperative programming language can be expressed as a program in a declarative language. Not only does this render the semantics executable, but it opens up the possibility of applying to imperative languages such as Java or C the advances made in program analysis and transformation of declarative languages. However the direct application of this approach returns the results of analysis and transformation in the language of the semantics. The problem is to provide a systematic way to relate the results back to the original imperative program.

In this paper a method is proposed for carrying out partial evaluation of imperative programs, using a partial evaluator for a declarative language, but returning the results in the syntax of the imperative program which is to be partially evaluated. The approach uses folding to reconstruct a specialised imperative program from a partially evaluated semantics-based interpreter.

The method provides a framework for constructing a partial evaluator for any imperative programming 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 folding mechanism for the declarative language. The correctness of the partial evaluator for the imperative language follows from the correctness of the partial evaluator and folding transformer of the declarative language. Other semantics-based program manipulations are suggested for imperative languages, based on abstract interpretation tools developed for declarative languages.

1 Introduction

Program semantics have long been used as a formal basis for program manipulation. By this is meant that the formal semantics of a programming language is written down in some suitable mathematical notation, which is then used to establish program properties such as termination, correctness with respect to specifications, or the correctness of program transformations. Declarative programming appears to be an elegant paradigm

for imperative language semantics implementation, giving an executable form of the semantics, and opening up the possibility of applying well-developed techniques for transformation and analysis of declarative languages to imperative languages as well.

Some attention has been devoted to partial evaluation of imperative programs, but each language requires specific techniques targeted to its particular syntax and constructs. Partial evaluation for declarative languages, on the other hand, has evolved considerably and there is a large literature, and some useful tools, for partial evaluation of logic and functional languages. This suggests the use of partial evaluation techniques in declarative languages, via semantics-based interpreters, for imperative program specialisation. The problem to be overcome is to return the results in the syntax of the imperative languages, otherwise the specialisation is useless to the programmer who wrote the imperative program to be specialised.

The experiments reported in this paper use logic programs, in particular constraint logic programs (CLP), for implementing the semantics of an imperative programming language. When written carefully, the semantics can be regarded as an interpreter for the imperative language. After specialising such an interpreter with respect to an input imperative program, the result is, in effect, a translation of the imperative program into CLP. We can also specialise further with respect to some input data for the imperative program. We may then apply abstract-interpretation-based analysis tools to obtain information about the behaviour of the imperative program. However, it is not immediately clear how to relate the results of such specialisation and analysis back to the original program.

Because reading the residual imperative program from the specialised interpreter can be troublesome, we have resorted to the folding rule of logic programs to go back to a more readable form of the specialised interpreter and thus a transformed version of the input imperative program.

Figure 1 depicts the steps proposed to achieve specialisation of imperative languages. Starting from a semantics-based interpreter, I, of an imperative language, we specialise the interpreter to an input imperative object program p and some partial input for p. This yields as a result another interpreter, I_p, where the interpretation layer has been removed by the partial evaluator. The representation of p and its input data is contained in I_p. Next, by means of folding we can aim at obtaining another equivalent version of I_p, say I'_p. I'_p contains the representation of a different imperative program, say p', constructed during the folding process, which we presume is the specialisation of the input imperative program p' with respect to its input data.

Here we report on our experience with using constraint logic programming, and logic programming program transformation techniques to achieve program transformation of imperative programs by means of a semantics-based interpreter of an imperative language.

Section 2 provides some remarks on the implementation of the operational semantics for a small imperative language resembling Pascal. In Section 3 we review some rules for partial evaluation. Section 4 gives an example of the method explored. Section 5 mentions methods of related work. Finally in Section 6 we state our final remarks, and some trends for future work.

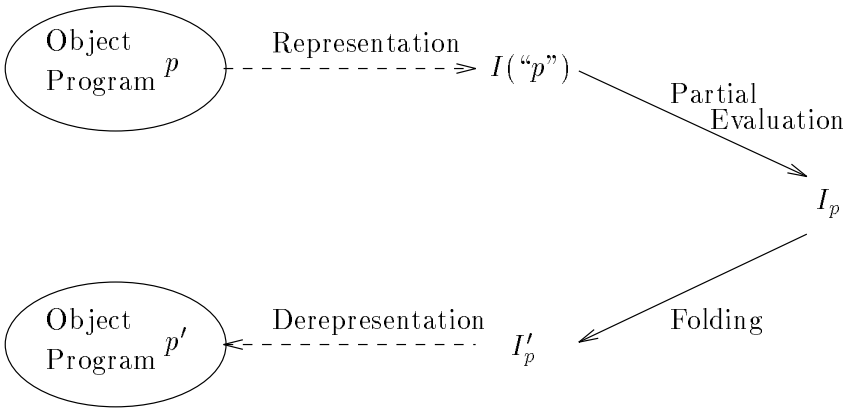


Figure 1: Imperative program transformation

2 Semantics-based Interpreter

Here we present briefly how we obtained an interpreter from operational semantics definitions, using constraint logic programming.

Assume we have an imperative language L for assignments, with arithmetic expressions, while statements, if-then-else conditionals, and boolean expressions. Let S_1, S_2 be two nonempty sequences of syntactically correct statements in L. Let a_1, a_2 be arithmetic expressions in L, b a boolean expression, s a variable environment (mapping variables to their value), and x a variable. For the semantic function $\mathcal{S}$ giving meaning to statements in L, the meanings of an assignment statement and composition of statements are:

$$\begin{aligned} \mathcal{S}[\![x := a]\!]s &= s[x \mapsto \mathcal{A}[\![a]\!]s] \\ \mathcal{S}[\![S_1; S_2]\!] &= \mathcal{S}[\![S_2]\!] \circ \mathcal{S}[\![S_1]\!] \end{aligned}$$

respectively, where $\mathcal{A}$ is the semantics function for arithmetic expressions. We represent the imperative program as a list of ground terms, each term being a statement. The above semantic clauses can be implemented as follows, where `assign(X,A)` represents an assignment statement `X := A`, and statement composition is represented using list notation, where `[S1|S2]` represented the composition of `S1` with the list of statement(s) `S2`.

```
statement(E1,E2,assign(X,Ae)) <- a_expr(E1,Ae,V),
                                 update(E1,E2,X,V)
statement(E,E,[]) <-
statement(E1,E3,[S1|S2]) <- statement(E1,E2,S1),
                            statement(E2,E3,S2)
```

`statement(E1,E2,P)` holds if for input variable environment `E1` the execution of program `P` in language L yields variable environment `E2`. The constraint used to define the `a_expr` predicate offers the possibility of enhancing partial evaluation of the interpreter

derived from the semantics, when implemented as follows for addition of arithmetic expressions:

```
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.

In order to implement the semantics in a constraint logic programming language, thus yielding an interpreter for the imperative language, we made several assumptions. Due to the lack of destructive assignment in logic languages we cannot have one logic variable for each imperative variable. This forces us to have a fresh list of logic variables each time we interpreted an imperative statement. The values of the imperative variables that do not change from execution of one statement to the following one are copied in the new environment variable. To implement the semantics function for arithmetic expressions, $\mathcal{A}$, we can invoke the constraint solver for integers.

3 Partial Evaluation

Partial evaluation generates a *residual program* by *evaluating* a program with *part* of its input which is available at compile-time. The aim of partial evaluation is to gain speed by specialising the source program exploiting the input available. Residual programs obtained by partial evaluation share some properties with the program from which they originated. We are interested in transformations which preserve the operational semantics. We will transform logic programs, and as a result we will obtain a transformed imperative program.

Here we recall some rules for logic program transformation, and in the next section we explain when to apply these rules. As a result of applying one of the following rules we generate a sequence of equivalent programs. Starting from an initial program, say P_0, and then applying some transformation rules each time producing another program P_i $(i > 0)$. Thus, the sequence $P_0, P_1, \ldots, P_n$ of programs. We want the final program, P_n, to have the same meaning as the first one, P_0. Then we can say that $\mathrm{Sem}(P_0) = \mathrm{Sem}(P_n)$ for a given semantics function Sem. For us, the semantics function used will be the computed answer with respect to the specialisation query. In order to achieve this we will make use of rules which are semantics preserving. For instance, the more specific version of a program [MNL88], the goal replacement rule , unfolding, and folding rules as presented in [PP94].

There are two basic transformation rules: *unfold* and *fold*. The former replaces the atom in the body of a clause by an instance of the body of a clause whose head unify with the atom to be replaced. This rule can be regarded as the macro expansion of imperative programming languages. The latter consists in replacing a list of atoms in the body of a clause by an instance of the head of another clause whose body is an instance of the list of atoms to be replaced. In addition to the above mentioned rules we need the *goal replacement* rule and a *more specific version* of a goal. Next we state shortly the rules as how to be applied. For a thorough explanation see [MNL88, PP94].

Let $A, E, D, R, C_k, E_j, D_i, C = H \leftarrow F, S, G$ be clauses, F, S, T,and G any sequence of atoms; θ_l and θ are substitution as required.

If we *unfold* C with respect to S using $D_1, \ldots, D_n$ in $P_j = \langle E_1, \ldots, E_r, C, E_{r+1}, \ldots, E_s \rangle$, we derive the clauses $C_1, \ldots, C_n$ and we get the new program $P_{k+1} = \langle E_1, \ldots, E_r, C_1, \ldots, C_n, E_{r+1}, \ldots, E_s \rangle$. S unifies with the head of each $D_1, \ldots, D_n$ with mgus $\theta_1, \ldots, \theta_n$ and each C_i has the form $(H \leftarrow F, \mathrm{bd}(D_i), G)\theta_i$.

If we *fold* C with respect to $\mathrm{bd}(D)\theta$ using D in P_j, we derive the clause $H \leftarrow F, \mathrm{hd}(D)\theta, G$, call it E, and we get a new program P_{k+1} by replacing C by E in P_k. $S = \mathrm{bd}(D)\theta$. And the variables which appear in $\mathrm{bd}(D)$ and not in $\mathrm{hd}(D)$ are renamed by θ and map to different variables not occurring in E.

By *replacement of* S *in* C *using* $S \equiv T$ we derive the clause $H \leftarrow F, T, G$, call it R, and we get P_{k+1} by replacing C by R in P_k.

By replacing a goal S by a *more specific version* $S\theta$ in clause C of program P_k we obtain the clause $(H \leftarrow F, S, G)\theta$, say E, and we get a new program P_{k+1} by replacing C by E in P_k. The justification is that all successful derivations of $P_k \cup \{\leftarrow S\}$ have computed answers of the form $\theta_1 = \theta\theta_2$.

4 Specialisation

In this section we show an example of program specialisation achieving constant propagation. This is a very simple example of partial evaluation but illustrates the method.

Through partial evaluation and folding we are going to derive a specialised program. For ease of presentation we assume that the input environment contains just the variables used in the example.

Example For the input environment, and input program:
$\langle [w, x, y, z], [W, X, Y, Z] \rangle$[1]

```
x := 3;
y := x*y;
z := x+1;
w := y+2;
if z =< x then    y := 1;
          else    y := 2;
```

We obtain a specialised program:

```
x :=3;
y := 3*y;
z := 4;
w := y+2;
y := 2;
```

[1]We use angle brackets instead of square brackets for ease of reading, to differentiate variable environments from lists of imperative/logic variables.

Assume that we have a semantics-based interpreter as outlined in Section 2. Then we partially evaluate the interpreter with respect to the following query which contains a representation of the original imperative program.

$$\leftarrow statement(\langle[w,x,y,z],A]\rangle, B, [assign(x,3), assign(y,times(x,y)), assign(z,plus(x,1)),\\ assign(w,plus(y,2)), ifte(le(z,x),[assign(y,1)],[assign(y,2)])])$$

By partial evaluation and more specific goal, to propagate further the variable environment to other subgoals in the partial evaluation process, we can derive the following specialised clause:

$$\begin{array}{l} statement(\langle[w,x,y,z],[W,X,Y,Z]\rangle, \langle[w,x,y,z],[W',3,2,4]\rangle,\\ \qquad [assign(x,3), assign(y,times(x,y)), assign(z,plus(x,1)),\\ \qquad assign(w,plus(y,2)), ifte(le(z,x),[assign(y,1)],[assign(y,2)])]) \leftarrow\\ \quad statement(\langle[w,x,y,z],[W,X,Y,Z]\rangle, \langle[w,x,y,z],[W,3,Y,Z]\rangle, assign(x,3)),\\ \quad \underline{a\_expr(\langle[w,x,y,z],[W,3,Y,Z]\rangle, x, X1),}\\ \quad a\_expr(\langle[w,x,y,z],[W,3,Y,Z]\rangle, y, Y1), \{Y' = X1 * Y1\},\\ \quad update(\langle[w,x,y,z],[W,3,Y,Z]\rangle, \langle[w,x,y,z],[W,3,Y'',Z]\rangle, y, Y'),\\ \quad \underline{a\_expr(\langle[w,x,y,z],[W,3,Y'',Z]\rangle, plus(x,1), 4),}\\ \quad update(\langle[w,x,y,z],[W,3,Y'',Z]\rangle, \langle[w,x,y,z],[W,3,Y'',4]\rangle, z, 4)\\ \quad statement(\langle[w,x,y,z],[W,3,Y'',4]\rangle, \langle[w,x,y,z],[W'',3,Y',4]\rangle, assign(w,plus(y,2))),\\ \quad statement(\langle[w,x,y,z],[W',3,Y',4]\rangle, \langle[w,x,y,z],[W',3,2,4]\rangle, assign(y,2)),\\ \quad statement(\langle[w,x,y,z],[W',3,2,4]\rangle, \langle[w,x,y,z],[W',3,2,4]\rangle, []) \end{array}$$

In this clause we also apply a goal replacement step to the underlined subgoals, followed by folding, with the clauses of the interpreter, obtaining the clause:

$$\begin{array}{l} statement(\langle[w,x,y,z],[W,X,Y,Z]\rangle, \langle[w,x,y,z],[W',3,2,4]\rangle,\\ \qquad [assign(x,3), assign(y,times(x,y)), assign(z,plus(x,1)),\\ \qquad assign(w,plus(y,2)), ifte(le(z,x),[assign(y,1)],[assign(y,2)])]) \leftarrow\\ statement(\langle[w,x,y,z],[W,X,Y,Z]\rangle, \langle[w,x,y,z],[W',3,2,4]\rangle,\\ \qquad [assign(x,3), assign(y,times(3,y)), assign(z,4),\\ \qquad assign(w,plus(y,2)), assign(y,2)] \end{array}$$

Using partial evaluation and folding we have achieved constant propagation for a small imperative language. Through partial evaluation of the constraints arising from the arithmetic operations we can propagate values through the program. Then we fold back the propagated values into the terms representing the imperative statements.

The effect of folding is to construct statements that have the same effect as the original ones.

5 Related Work

Here we will consider work on partial evaluation of imperative languages. There has been important work [JGS93, CD91] on the specialisation of imperative languages denotational definitions to functional languages. Nevertheless we will only discuss the work which focusses on source-to-source specialisation of imperative languages.

5.1 Pascal specialisation

In [Mey91] a technique for partial evaluation of programs in a subset of the Pascal programming language is presented. Transformations in their method are coded as rules for transformations for each construct in Pascal. The partial evaluator is coded in Lisp. There is no clear separation between the semantics representation (semantics-based interpretation/execution), and the partial evaluation. Thus the correctness of their method is taken for granted. We believe our approach to partial evaluation provides a better understanding of the process of constructing a partial evaluator for imperative programs. Constant propagation can be achieved using this method, similar to what we show in this paper.

5.2 Ada specialisation

A subset of the Ada programming language can be simplified or specialised using the authors' method called *symbolic execution* [CPPGM91]. Symbolic execution resembles partial evaluation in the sense that execution of the program on a partial input executes as much as possible on the input program. Thus generating a so called residual program which executes the same as the input program when given the whole input, and the residual program is given the remaining input. Two functions in an operational semantics style define partial evaluation for any program. A prototype of this system was implemented in Lisp. From the paper is not clear how to achieve constant propagation as we propose in this paper. The authors argue that it can be described in the functions *exec* and *simpl* for assignment, but no further mention is made in the paper. Instead, conventional data-flow analysis techniques [ASU86] are applied to the program to achieve constant and copy propagation. Hence the symbolic executor by Coen et al. presumably does not propose a new method to achieve constant propagation.

5.3 Fortran specialisation

Kleinrubatscher et al. [KKZG95] describe partial evaluation for a subset of the Fortran 77 programming language. Partial evaluation is a 4-step process: Translation from the Fortran program into a low level code (Core Fortran); monovariant programs variable binding-time analysis and annotation of the program statements as static or dynamic; specialisation of the annotated low level code; and translation into the source language, Fortran 77. The partial evaluator was written in Fortran. This approach to partial evaluation resembles that of generating extensions as in [And94].

Constant propagation is possible with this approach, but no live/dead analysis was achieved. The monovariant binding-time analysis results in little constant propagation. The correctness of the partial evaluation method used appears to be based on the correctness of the methods presented in [ASU86].

5.4 C specialisation

Analysis and transformation of programs in ANSI C was the target of Andersen's work [And94]. The specialiser is automatic and generates generating extensions in C.

The C program specialisation is achieved in several steps. Using a monovariant variable binding-time analysis the source C program is analysed. The analysed C program is transformed thus producing a generating extension program. Attaching the appropriate libraries and the partial input, the generating extension is executed. The output of such an execution is a specialised C program. The set of C libraries describe the transformations that the generated extension commands. The correctness of the transformations which generate extensions are presumably based on correctness of executing the programs (generating extensions) using a C compiler. That is, all the transformations are carried out in the C programming language. Any semantic mismatch between the source program and the specialised one, can be attributed to the C compiler used and to the correctness of the generating extensions functions implemented in the libraries.

5.5 Hard Real Time Languages

In [NP92] the author describes a technique to partially evaluate imperative programs for hard real-time applications. Hard real-time programs must obey several restrictions on the program behaviour such as predictability. That is, no infinite loops can occur. For this to be possible the authors enforce that the compile-time store remains unchanged when a guard in a loop or conditional statement is not compile-time. This statement suggests that using this method only simple constants can be propagated. Simple constants are all values that can be proved to be constant subject to two constraints: no information is assumed about which direction branches will take, and only one value for each variable is maintained along each path in the program [WZ85].

Evidential proof is provided through 3 examples. The authors propose a language as well as a partial evaluator. Unable to asses whether they can do constant propagation or not.

Although most of the contributions are labelled as techniques for specialisation of imperative languages, only one language is explored in each paper, and it is not clear how to extend the techniques to other languages. Besides, we believe the methods above do not commit to a semantics-based approach. None of them take the semantics of the language studied as a basis for transforming the language programs, but consider the semantics as a correctness proof aid, as in [NP92].

6 Concluding remarks

Using a description of the operational semantics of a subset of an imperative language we implemented a logic program which mimics the semantics. That logic program can be regarded as an interpreter for the imperative language considered. With the help of a partial evaluator for constraint logic programs we can obtain program specialisation of imperative programs. Imperative programs so specialised end up as specialised logic programs, whereas what is needed is a specialised program in the imperative source language. We have resorted to specialisation taking advantage of the logic programming program transformation operations (fold, unfold, etc.), but generating code that is very close to the source language. That is, no translation from an imperative language to a declarative language is performed. Thus, we guarantee correctness of the transformations by correctness of the partial evaluator and folding. As the interpreter is based on the semantics of the imperative language we ensure semantics preserving transformations.

We began this work expecting to develop a framework for semantics-based partial evaluation of imperative languages. We claim to have a general method based on the semantics of an imperative language, a partial evaluator, and a program transformer based on folding and goal replacement.

Not all imperative language constructs share a common semantics definition. There are subtle differences for each language, for each construct. Nevertheless, we believe that the technique, proposed here, can be generalised to more imperative languages, provided we had implemented the semantics definitions for the desired language.

Future work. Currently we are working to find an automatic way for performing constant propagation. That is, include a module for folding and unfolding in an existing partial evaluator to reach our aims in constant propagation. The semantics of procedures have been successfully implemented in Prolog. We seek strategies for unfolding and folding procedure calls. We plan to extend the code which we have for procedures to cope with different parameter passing styles. Blocks with dynamic and static scope of variables and procedures have been implemented as well. Static scope of variables was implemented using locations, and multidimensional arrays are also handled. More complex data structures remain to be explored, strings for instance, and user-defined data structures. Division, exception handling, object-oriented features and input/output are other challenging issues which we are studying.

References

[And94] Lars Ole Andersen. *Program Analysis and Specialization for the C programming Language*. PhD thesis, DIKU, University of Copenhagen, Denmark, May 1994.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.

[CD91] Charles Consel and Olivier Danvy. Static and dymanic semantic processing. In ACM Press, editor, *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 14–24, Orlando, Florida, 1991.

[CPPGM91] Alberto Coen-Porisini, Flavio De Paoli, Carlo Ghezzi, and Dino Mandrioli. Software specialization via symbolic execution. *IEEE Transactions on Software Engineering*, 17(9):884–899, 1991.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International Series, 1993.

[KKZG95] Paul Kleinrubatscher, Albert Kriegshaber, Robert Zöling, and Robert Glück. Fortran program specialization. *SIGPLAN Notices*, 30(4):61–70, 1995.

[Mey91] Uwe Meyer. Techniques for partial evaluation of imperative programs. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 94–115, New Haven, Conneticut, 1991. ACM Press.

[MNL88] K. Marriot, L. Naish, and J.-L. Lassez. Most specific logic programs. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, volume 2, pages 909–923, Washington, Seattle, 1988. MIT Press.

[NP92] Vivek Nirke and William Pugh. Partial evaluation of high-level imperative programming languages, with applications in hard real-time systems. In ACM Press, editor, *Conference Record of the Nighteenth Symposium on Principles of Programming Languages*, pages 269–280, Albuquerque, New Mexico, 1992.

[PP94] Alberto Pettorossi and Maurizio Proietti. Transformations of logic programs: Foundations and techniques. *Journal of Logic Programming*, 19(20):261–320, 1994.

[WZ85] Mark N. Wegman and Frank Kenneth Zadeck. Constant propagation with conditional branches. In *Conference Record of the Twelfth Symposium on POPL*, pages 291–299, New Orleans, Luisiana, 1985.