Equations as a Uniform Framework for Partial Evaluation and Abstract Interpretation

J. Field, J. Heering, T.B. Dinesh

Software Engineering (SEN)

SEN-R9722 November 30, 1997

Report SEN-R9722
ISSN 1386-369X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

Equations as a Uniform Framework for Partial Evaluation and Abstract Interpretation

J. Field
IBM T. J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598, USA

J. Heering, T.B. Dinesh
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

ABSTRACT
A variety of disparate methods have traditionally been used to define the execution semantics of programming languages, to describe partial evaluation, to formalize program analysis as abstract interpretation, and to implement each of these operations in practical systems. We argue here that *equational logic* can serve to unify each of these aspects of language manipulation.

1991 Computing Reviews Classification: D.3.4, F.3.2

Keywords and Phrases: equational logic, algebraic specification, program analysis, partial evaluation, abstract interpretation

Note: Submitted to the SOPE'98, special issue of the ACM Computer Surveys. This research was supported in part by the Netherlands Organization for Scientific Research (NWO) under the *Generic Tools for Program Analysis and Optimization* project.

1. Introduction

A variety of disparate methods have traditionally been used to define the execution semantics of programming languages, to describe partial evaluation, to formalize program analysis as abstract interpretation, and to implement each of these operations in practical systems. We argue here that *equational logic* can serve to unify each of these aspects of language manipulation. The benefits of such a unified view are considerable:

- Specifications of interpreters, partial evaluators, or analyzers can often be done in a *modular* fashion, simply by adding or removing sets of equations.
- Equational systems are particularly amenable to efficient mechanical implementation, using such techniques as Knuth-Bendix completion [Dershowitz and Jouannaud 1990], term graph rewriting [Kennaway et al. 1994], and compilation to C [Borovanský et al. 1996; Didrich et al. 1994; Kamperman and Walters 1996].
- Equational semantics is very well understood. A rich body of notions and results can be brought to bear to prove useful properties of equational systems, such as completeness, soundness, and relations between various models [Meseguer and Goguen 1985; Wirsing 1990].
- Implementations and formal results used for one application of equational semantics (e.g., interpreters) can often be reused or extended with ease to other settings (e.g., partial evaluators).

```
program
do  $\overline{x} := \overline{a}$;  $\overline{w} := \overline{x}$;  $\overline{w}\uparrow := \overline{b}$;
    if  $\overline{w}\uparrow\uparrow = \overline{x}\uparrow$
        then  $\overline{y} := \overline{x}\uparrow$
        else  $\overline{y} := \overline{c}$
in  $\overline{y}\uparrow$
```

Figure 1: Example **P** Program.

The authors are currently engaged in an effort to show that equational logic can be beneficially applied to the analysis and transformation of "industrial-strength" imperative languages such as C. Some promising initial steps in this direction are detailed in [Bergstra et al. 1997; Field 1992], in which a graph-based language-neutral *intermediate representation* called PIM and a collection of related equational systems are defined. In this article, we will describe various properties and applications of equational systems informally, using as a running example a programming language that is a simplified cousin of PIM. Readers interested in formal definitions of the ideas discussed here, applied in the context of a richer equational system, should refer to [Bergstra et al. 1997]. For another example of an equational approach to defining the semantics of an imperative language, see [Hoare et al. 1987].

2. Equational Interpreters

Consider the program in Figure 1, written in a tiny imperative programming language, **P** (originally defined in [Field and Tip 1994]). A **P do** construct is executed by evaluating its statement list, and using the computed values to evaluate its **in** expression. (Note it is *not* a loop construct, but executed only once.) Expressions of the form '$\overline{x}$' are atoms, and play the dual role of basic values and addresses which may be assigned to using '$:=$'. Addresses are explicitly dereferenced using '$\uparrow$'; an address dereferenced more than once behaves like a pointer. The distinguished atoms $\overline{t}$ and $\overline{f}$ represent boolean values.

Now consider the equations in Figure 2. Capitals denote variables whose value is an arbitrary construct of the appropriate type. Hence, the right- and left-hand sides of equations are **P** program fragments. Viewed separately, each equation defines a natural property of a **P** program fragment as a semantics-preserving program transformation. Viewed collectively, however, the equations can function as an interpreter in which each equation is oriented from left to right to form a *rewriting rule*. Such rules are then applied to a matching *redex* fragment of a **P** program until no further equations are applicable. The *normal form* program that results from evaluating the program in Figure 1 is $\overline{b}$.

A rewriting system such as the one in Figure 2 is often called a *calculus*. To define a calculus, one must ensure that there are sufficient equations to normalize every closed program, that normal forms are indedependent of the order in which equations are applied (assuming a deterministic semantics is desired), and that some orientation of the equations and strategy for choosing the next redex exists which ensures that the rewriting of all terminating programs terminates. In particularly well-behaved systems (such as the one in Figure 2), any strategy for choosing redexes will eventually yield a normal form.

3. Partial Evaluation from an Equational Perspective

The equations of Figure 2 are sufficient to normalize every closed **P** program, but the rules of equational logic are such that they can be applied to *open* **P** programs as well. Apart from the limited features of **P**, such programs contain missing inputs, unbound arguments, or even missing statements, and are similar to the program fragments making up the right- and left-hand sides of the equations in Figure 2. Hence, a very natural and general formulation of partial evaluation in an equational setting is simply *the rewriting of open programs using semantics-preserving equational transformation*[1] [Heering 1986]. However, while an equational operational

[1]This formulation is not always the right one in settings in which partial evaluators must be *self-applicable* [Jones et al. 1993]. However, self-application is not required in many optimization applications.

[P1]	$\overline{a} = \overline{a}$	$=$	$\overline{t}$	for all constants a
[P2]	$\overline{a} = \overline{b}$	$=$	$\overline{f}$	for all constants a, b such that $a \neq b$
[P3]	$\textbf{if } \overline{t} \textbf{ then } E_1 \textbf{ else } E_2$	$=$	E_1	
[P4]	$\textbf{if } \overline{f} \textbf{ then } E_1 \textbf{ else } E_2$	$=$	E_2	
[P5]	$\textbf{if } \overline{t} \textbf{ then } S_1 \textbf{ else } S_2$	$=$	S_1	
[P6]	$\textbf{if } \overline{f} \textbf{ then } S_1 \textbf{ else } S_2$	$=$	S_2	
[P7]	$\textbf{do } S \textbf{ in } \overline{X}$	$=$	$\overline{X}$	
[P8]	$\textbf{do } S \textbf{ in } E_1 = E_2$	$=$	$(\textbf{do } S \textbf{ in } E_1) = (\textbf{do } S \textbf{ in } E_2)$	
[P9]	$\textbf{do } S \textbf{ in if } E_1 \textbf{ then } E_2 \textbf{ else } E_3$	$=$	$\textbf{if } (\textbf{do } S \textbf{ in } E_1)$	
			$\textbf{then } (\textbf{do } S \textbf{ in } E_2)$	
			$\textbf{else } (\textbf{do } S \textbf{ in } E_3)$	
[P10]	$\textbf{do } E_1 := E_2 \textbf{ in } (E_3 \uparrow)$	$=$	$\textbf{if } (\textbf{do } E_1 := E_2 \textbf{ in } E_3) = E_1$	
			$\textbf{then } E_2$	
			$\textbf{else } ((\textbf{do } E_1 := E_2 \textbf{ in } E_3) \uparrow)$	
[P11]	$\textbf{do } S; E_1 := E_2 \textbf{ in } (E_3 \uparrow)$	$=$	$\textbf{if } (\textbf{do } S; E_1 := E_2 \textbf{ in } E_3) = (\textbf{do } S \textbf{ in } E_1)$	
			$\textbf{then } (\textbf{do } S \textbf{ in } E_2)$	
			$\textbf{else do } S \textbf{ in } ((\textbf{do } E_1 := E_2 \textbf{ in } E_3) \uparrow)$	
[P12]	$\textbf{do if } E \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ in } E$	$=$	$\textbf{if } E \textbf{ then } (\textbf{do } S_1 \textbf{ in } E) \textbf{ else } (\textbf{do } S_2 \textbf{ in } E)$	
[P13]	$\textbf{do } S_1; \textbf{if } E_1 \textbf{ then } S_2 \textbf{ else } S_3 \textbf{ in } E_2$	$=$	$\textbf{if } (\textbf{do } S_1 \textbf{ in } E_1)$	
			$\textbf{then } (\textbf{do } S_1; S_2 \textbf{ in } E_2)$	
			$\textbf{else } (\textbf{do } S_1; S_3 \textbf{ in } E_2)$	
[P14]	$\textbf{program } \overline{X}$	$=$	$\textbf{result } \overline{X}$	

Figure 2: Equations Defining an Interpreter for **P**.

[PF1]	$(\overline{x} := \overline{a};\ \overline{y} := \overline{b})$	$=$	$(\overline{y} := \overline{b};\ \overline{x} := \overline{a})$
[PF2]	$(\overline{x} := \overline{y};\ \overline{x}\uparrow := \overline{a})$	$=$	$(\overline{x} := \overline{y};\ \overline{y} := \overline{a})$

Figure 3: Some of the Equations in the Final Algebra Enrichment.

[PO1]	**if** P **then** X **else** X	$=$	X
[PO2]	**do** $E_1 := E_2$ **in** $(E_1\uparrow)$	$=$	E_2
[PO3]	**do** $E_1 := E_1$ **in** E_2	$=$	E_2

Figure 4: Some of the Equations in the ω-Complete Enrichment.

semantics such as that in Figure 2 is a useful starting point for partial evaluation, it is not usually sufficient and has to be enriched in various ways.

3.1 Final Algebra Enrichments

Consider Eq. **[PF1]** in Figure 3. Intuitively, it seems reasonable to view its right- and left-hand side as equivalent, since a program optimizer should be free to permute the assignments when appropriate. However, no combination of the rules in Figure 2 allows these fragments to be equated.

The concept not captured by the "operational" equations in Figure 2 is the notion of *observable equivalence*. The idea is simple: if the result of inserting two distinct closed program fragments in *all* result-yielding closed program contexts is the same, then the two fragments should be viewed as equivalent. The algebraic equivalent of observational equivalence is the notion of a *final algebra* [Meseguer and Goguen 1985, Sec. 5][Wirsing 1990, Sec. 5.4].

One pleasant property of equational semantics is that observable equivalence can often be captured simply by enriching the equations that define an operational, or *initial* semantics, with additional equations that axiomatize observational equivalence. For **P** such an enrichment would include equations such as **[PF1]** and **[PF2]**, although probably in a more general form.

3.2 ω-Complete Enrichments

However, to produce an equational system useful for partial evaluation, a final algebra enrichment is still not sufficient. Consider Eq. **[PO1]** in Figure 4, where P and X are arbitrary **P** fragments of the appropriate type. In this example, it is evident that Eq. **[PO1]** should hold. It is typical of the sort of transformation carried out during partial evaluation. The problem in this case is that not all valid equivalences on *open* programs are equational consequences of the operational set of equations.

An equational system in which all equations valid on open programs may be deduced using only equational reasoning is called *ω-complete* [Heering 1986]. Although full ω-completeness cannot always be achieved, we believe it is a very natural starting point for partial evaluation applied to program optimization and analysis. In such systems, no structural induction is needed to prove equivalences on open programs. Trading structural induction for equational reasoning from an enriched equational base has two potential advantages:

- An existing rewriting implementation of equational logic can be used to carry out partial evaluation, although an A-, AC-, or some other E-rewriting capability [Dershowitz and Jouannaud 1990] may be needed depending on the equational system involved.
- Rewriting may have a sense of direction lacking in structural induction. It may perform useful simplification of programs without having been given an explicit inductive proof goal beforehand.

Similar advantages apply in the case of the final algebra enrichment, but with *context induction* [Hennicker

1990] rather than structural induction being replaced by equational reasoning as the means for proving observational equivalence.

4. An Equational Approach to Abstract Interpretation

Abstract interpretation [Cousot and Cousot 1992] (and its cousin, dataflow analysis [Muchnick and Jones 1981]) defines a nonstandard semantics for a language that captures some set of program properties that are not revealed by the program's concrete semantics. We believe that equational systems can serve as a useful tool for defining abstract interpretations. Depending on the nature of the desired interpretation, several approaches are possible.

- Add new equations, leaving the existing term structure intact. Such an approach is often useful to eliminate distinctions made by the standard semantics that are "uninteresting" to the user, and which may also inhibit certain optimizations. Consider an extension of **P** in which nontermination can occur. One abstract interpretation of the extended version of **P** might include an equation of the form

  $$\textbf{[PA1]} \quad (X = X) \quad = \quad \bar{t},$$

  whereas the standard semantics would likely exclude such an equation, since it would be invalid if X were divergent. Nevertheless, adding an equation of this form would allow optimizations to be made that are innocuous, in the sense that they only introduce additional non-divergent results.

- Homomorphically map certain *ground* observable value terms to terms representing "abstract values" of interest (e.g., types or signs of integers), altering only those equations that make non-trivial use of such values (i.e., equations that do not hold for all values). The modular nature of equational systems make this easy. Using this approach, Dinesh [1996] has systematically derived the static semantics of an object-oriented language from its execution semantics. In the case of **P** *all* equations in Figure 2 would likely hold in any abstract interpretation of practical interest.

- Map certain *non-ground* terms to more complex terms that encode intensional aspects of a language; add equations that axiomatize the properties of those features, but which introduce *no new equivalences on observable values*. Consider, e.g., equation **[P13]** in Figure 2. This equation propagates the set of assignments that occur outside a conditional into each of the three arms of the conditional. A more conventional semantics would probably have first evaluated the conditional, then used the assignments in whatever expression the conditional yields. Taking this idea to an extreme, the equational system PIM defined in [Bergstra et al. 1997] includes terms that represent *reaching definitions*, i.e., assignments to a variable that may affect some subsequent use of that variable. The equivalences on ground terms, however, are exactly what one would expect in any imperative language. By partially evaluating PIM terms to appropriate non-normal forms, reaching definitions become self-evident from the term structure. In this manner, abstract interpretation is performed by partial evaluation of a *concrete* semantics. This sort of trick is particularly easy to pull off in equational semantics, since intermediate states of evaluation are "first-class" entities.

5. Implementing Equational Systems

After enrichment, the resulting equational systems tend to be much less amenable to a straightforward implementation as oriented rewriting systems than the original operational semantics definition they were obtained from. It is a distinct advantage of the equational framework that it is sufficiently general to allow enrichments that are hard to implement, but relatively easy to develop. This is made possible by the powerful implementation techniques mentioned in Sec. 1.

*Acknowledgments:*We are grateful to Jan Bergstra and G. Ramalingam for valuable technical discussions.

REFERENCES

BERGSTRA, J. A., DINESH, T. B., FIELD, J., AND HEERING, J. 1997. Toward a complete transformational toolkit for compilers. *ACM Trans. Program. Lang. Syst. 19,* 5 (September), 639–684.

BOROVANSKÝ, P., KIRCHNER, C., KIRCHNER, H., MOREAU, P.-E., AND VITTEK, M. 1996. ELAN: A logical framework based on computational systems. In *Proceedings of the First International Workshop on Rewriting Logic*, J. Meseguer, Ed. Vol. 4. Asilomar (California). Electronic Notes in Theoretical Computer Science.

COUSOT, P. AND COUSOT, R. 1992. Abstract interpretation and application to logic programs. *Journal of Logic Programming 13,* 103–179.

DERSHOWITZ, N. AND JOUANNAUD, J.-P. 1990. Rewrite systems. In *Handbook of Theoretical Computer Science, Vol. B, Formal Models and Semantics*, J. van Leeuwen, Ed. Elsevier/The MIT Press, 243–320.

DIDRICH, K., FETT, A., GERKE, C., GRIESKAMP, W., AND PEPPER, P. 1994. OPAL: Design and implementation of an algebraic programming language. In *International Conference on Programming Languages and System Architectures*, J. Gutknecht, Ed. Lecture Notes in Computer Science, vol. 782. Springer-Verlag, 228–244.

DINESH, T. B. 1996. A kernel object-oriented language. In *Language Prototyping: An Algebraic Specification Approach*, A. van Deursen, J. Heering, and P. Klint, Eds. AMAST Series in Computing. World Scientific, Chapter 3, 53–84.

FIELD, J. 1992. A simple rewriting semantics for realistic imperative programs and its application to program analysis. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*. San Francisco, 98–107. Published as Yale University Technical Report YALEU/DCS/RR–909.

FIELD, J. AND TIP, F. 1994. Dynamic dependence in term rewriting systems and its application to program slicing. In *Proceedings of the Sixth International Symposium on Programming Language Implementation and Logic Programming*, M. Hermenegildo and J. Penjam, Eds. Vol. 844. Springer-Verlag, 415–431.

HEERING, J. 1986. Partial evaluation and ω-completeness of algebraic specifications. *Theoretical Computer Science 43*, 149–167.

HENNICKER, R. 1990. Context induction: A proof principle for behavioral abstraction. In *Design and Implementation of Symbolic Computation Systems (DISCO '90)*, A. Miola, Ed. Lecture Notes in Computer Science, vol. 429. Springer-Verlag, 101–109.

HOARE, C., HAYES, I., JIFENG, H., MORGAN, C., ROSCOE, A., SANDERS, J., SORENSEN, I., SPIVEY, J., AND SUFRIN, B. 1987. Laws of programming. *Communications of the ACM 30,* 8 (August), 672–686. Corrigenda, ibid., p. 770.

JONES, N. D., GOMARD, C. K., AND SESTOFT, P. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall.

KAMPERMAN, J. F. T. AND WALTERS, H. R. 1996. Minimal term rewriting systems. In *Recent Trends in Data Type Specification (WADT '95),* M. Haveraaen, O. Owe, and O.-J. Dahl, Eds. Lecture Notes in Computer Science, vol. 1130. Springer-Verlag, 274–290.

KENNAWAY, J. R., KLOP, J. W., SLEEP, M. R., AND DE VRIES, F. J. 1994. On the adequacy of graph rewriting for simulating term rewriting. *ACM Trans. Program. Lang. Syst. 16,* 3, 493–523.

MESEGUER, J. AND GOGUEN, J. A. 1985. Initiality, induction and computability. In *Algebraic Methods in Semantics*, M. Nivat and J. C. Reynolds, Eds. Cambridge University Press, 459–541.

MUCHNICK, S. S. AND JONES, N. D., Eds. 1981. *Program Flow Analysis : Theory and Applications*. Prentice-Hall.

WIRSING, M. 1990. Algebraic specification. In *Handbook of Theoretical Computer Science, Vol. B, Formal Models and Semantics*, J. van Leeuwen, Ed. Elsevier/The MIT Press, 675–788.