ROPAS Research On Program Analysis System
National Creative Research Initiative Center
Korea Advanced Institute of Science and Technology

ROPAS MEMO
2001-10
June 28, 2001

Yet Another Ensemble of Abstract Interpreter, Higher-Order Data-Flow Equations, and Model Checking *

Kwangkeun Yi
http://cs.kaist.ac.kr/~kwang
Dept. of Computer Science
Korea Advanced Institute of Science and Technology

March 2, 2001

1 This Work

This work is a first step to our goal: building a "realistic" program analysis generator, like lex and yacc in syntax analysis. Toward this goal one question is which program analysis technique(s) should be supported by such a tool?

We agree that with a right cooperation of different techniques might we be able to build a realistic tool. We are currently building a program-analysis generator (named "Zoo") that is an ensemble of three frameworks: abstract interpretation[CC77, CC92], conventional data-flow analysis[RP86, KU76, KU77], and model checking[CGP99]. The Zoo system applies each of the three frameworks to a different phase in analysis' specification and generation.

A similar combinational methodology has been presented by Schmidt and Steffen[SS98]. The main difference of Zoo is in how to build finite models of the input programs. Given an input program, Zoo compiles it into a set of data-flow equations by referencing the abstract interpreter specification. On the way to reach the solutions of the equations, a finite data-flow graph is constructed, which is later queried by model checking for program properties of interest. Moreover, the abstract interpreter is in the style of the definitional interpreter[Rey98a, Rey98b, Rey72], not in the style of a small-step operational semantics. This stylistic difference is meaningful in an analysis generation tool, because small-step operational semantics can be too fine-grained (hence, low-level) semantic specifications for higher-order languages.

In Zoo, each of the three frameworks plays the following role:

- The user (analysis designer) designs a program analysis in the abstract interpretation framework[CC77, CC95]. Abstract interpreter is represented in the style of an definitional interpreter[Rey98a, Rey98b, Rey72], which is usually a recursive function. The user is expected to prove its correctness.
- Given an input program to analyze, instead of interpreting it by the abstract interpreter function as in [YH93], Zoo compiles the input program into a set of

*This work is supported by Creative Research Initiatives of the Korean Ministry of Science and Technology.

data-flow equations by referencing the abstract interpreter. The unknowns in the equations are the program's approximate states at each of the program's control points.

- Then Zoo solves the derived data-flow equations. Here, it can use the (iteration or elimination) algorithms[Hec77, RP86, KU76, DGS94, MR90, RP86, Ros80, GW76, Tar76] in the conventional data-flow analysis framework. When the equations are higher-order[1], which is when the input program is higher-order (functions as first-class objects), it uses the iterative fixpoint methods[Jør93, CHY95, Bou93] developed in general abstract interpretation framework.

- The computed solution is considered a finite model of the program. The model is a data-flow graph whose nodes (the solutions of the equations) are the states at program points and the edges are paths where the states flow. This graph is constructed while Zoo solves the equations.

- Extracting an information (or, querying a program property) from the finite program-model is done by model checking[CGP99]. Zoo provides a language for the computational tree logic (CTL)[CGP99, BCM$^+$92, CE81]. The complexity of the CTL checker is linear to the sizes of the model and the formula.

2 Abstract Interpreter Into Data-Flow Equations

2.1 First-Order Imperative Language

Consider an imperative language:

$$
\begin{array}{lclll}
\mathit{Expr} & e & ::= & z \mid x & \text{integer/variable} \\
 & & \mid & e_1 \;\texttt{+}\; e_2 & \text{primitive operation} \\
 & & \mid & x \;\texttt{:=}\; e & \text{assignment} \\
 & & \mid & e \;\texttt{;}\; e & \text{sequence} \\
 & & \mid & \texttt{if}\; e_1\; e_2\; e_3 & \text{choice}
\end{array}
$$

Let's assume that we are interested in the sign (negative, zero, or positive) of the program variables. An abstract interpreter

$$E \in \mathit{Expr} \times \mathit{State} \to \mathit{Sign} \times \mathit{State}$$

can be defined as follows:

$$s \in \mathit{State} = \mathit{Var} \to \mathit{Sign}$$

[1]The index of a variable in the equations is unknown.

$$
\begin{array}{llcl}
\text{(a)} & E(z,s) & = & (sign(z), s) \\
\text{(b)} & E(x,s) & = & (s(x), s) \\
\text{(c)} & E(x\texttt{:=}e,s) & = & let\ (v_1,s_1) = E(e,s) \\
 & & & in\ (v_1, s_1[v_1/x]) \quad \text{(* state update *)} \\
\text{(d)} & E(e_1\texttt{;}e_2,s) & = & let\ (v_1,s_1) = E(e_1,s) \\
 & & & \quad (v_2,s_2) = E(e_2,s_1) \\
 & & & in\ (v_2,s_2) \\
\text{(e)} & E(e_1\texttt{+}e_2,s) & = & let\ (v_1,s_1) = E(e_1,s) \\
 & & & \quad (v_2,s_2) = E(e_2,s_1) \\
 & & & in\ (add(v_1,v_2), s_2) \\
\text{(f)} & E(\texttt{if}\ e_1\ e_2\ e_3,s) & = & let\ (v_1,s_1) = E(e_1,s) \\
 & & & \quad (v_2,s_2) = E(e_2,s_1) \\
 & & & \quad (v_3,s_3) = E(e_3,s_1) \\
 & & & in\ (v_2,s_2) \sqcup (v_3,s_3)
\end{array}
$$

where the state s is a finite table from program variables to signs, $sign(z)$ determines the sign of an integer z, $add(v_1, v_2)$ is an addition over the sign domain[2]. The correctness of the above abstract interpreter, which is to be checked by the user, is to prove that the meaning of the recursive abstract interpreter function contains that of concrete one. In general, the meaning of a recursive function $f \in A \to B$ is defined as the least fixpoint $lfp F$ of its corresponding functional $\mathcal{F} \in (A \to B) \to A \to B$ (by interpreting the domains as complete-partial-order sets), where $\mathcal{F}(f)$ is identical as the body of f. Thus, letting a concrete functional be $\mathcal{F}$ and an abstract one $\mathcal{E}$, the user has to prove

$$lfp\mathcal{F} \sqsubseteq lfp\mathcal{E}.$$

Now, given the correct abstract interpreter definition E and an input program, say,

$$\overbrace{\underbrace{\texttt{x := 1;}}_{1}\ \underbrace{\texttt{y := x+1}}_{2}}^{0}$$

(We indexed three expressions by 0,1, and 2.) Zoo compiles the program into a set of equations. The unknowns in the equations are the pairs of X_i^+ and X_i^- for each expression i. X_i^+ represents the pre-state right before evaluating expression i, and X_i^- the post-state right after evaluating expression i. The unknowns range over $State$. By referencing the interpreter E, the equations between those unknowns are derived as follows. Expression 0 is a sequence expression, hence it's the case (d) of E:

$$
\begin{array}{llcll}
\text{(d)} & E(e_1\texttt{;}e_2,s) & = & let\ (v_1,s_1) = E(e_1,s) & (*) \\
 & & & \quad (v_2,s_2) = E(e_2,s_1) & (**) \\
 & & & in\ (v_2,s_2) & (***)
\end{array}
$$

[2]The sign domain is a lattice whose elements have a sound relation (Galois connection[CC77]) with the powerset of integers. For example, "positive" relates to the set of positive integers or non-termination, "negative" to the set of non-positive integers or non-termination, "don't know" to the whole set of integers or non-termination, and "undefined" to the empty set of integers or non-termination.

From line $(*)$, the pre-state s to the sequence expression is equal to the pre-state to the first sub-expression e_1. Thus, in terms of the example program's expressions, pre-state to expression 0 is equal to the pre-state to expression 1:

$$X_1^+ = X_0^+.$$

From line $(**)$, the post-state s_1 from $E(e_1, s)$ is equal to the pre-state to the second sub-expression e_2, hence

$$X_2^+ = X_1^-.$$

From line $(***)$, the post-state s_2 from $E(e_2, s_1)$ is equal to the post-state of the original expression, hence

$$X_0^- = X_2^-.$$

Similarly, the equations for all other sub-expressions are derived by looking up the sub-expressions' cases in the abstract interpreter.

Note that solving such set of equations are done by the algorithms[Hec77, RP86, KU76, DGS94, MR90, RP86, Ros80, GW76, Tar76] developed in the conventional data-flow analysis.

2.2 Higher-Order Applicative Language

Consider a call-by-value higher-order applicative language. For presentation brevity, we assume the dynamic scoping:

$$\begin{array}{rcll} e & ::= & z & \text{integer} \\ & | & x & \text{variable} \\ & | & \lambda x.e & \text{abstraction} \\ & | & e_1\ e_2 & \text{application} \end{array}$$

Let's assume that we're interested only in lambda expressions $(\lambda x.e)$ as values (that is, integer values are completely abstracted). An abstract interpreter, again as a recursive function, is defined as

$s \in \mathit{State} = \mathit{Var} \to \mathit{Expr} + \{\bullet\}$ (* $\bullet$ stands for all integers or non-termination. *)

$$\begin{array}{lrcll} \text{(a)} & E(x, s) & = & s(x) & \\ \text{(b)} & E(\lambda x.e, s) & = & \lambda x.e & \\ \text{(c)} & E(e_1\ e_2, s) & = & \mathit{let}\ \lambda x.e' = E(e_1, s) & (*) \\ & & & \qquad v = E(e_2, s) & (**) \\ & & & \qquad v' = E(e', s[v/x]) & (***) \\ & & & \mathit{in}\ v' & \end{array}$$

Now, the set of equations compiled from an input program becomes higher-order. Consider an input program:

$$\overbrace{\underbrace{(\lambda x.x\ 1)}_{1}\,\underbrace{(\lambda y.y)}_{2}}^{0}$$

(We indexed three expressions by 0, 1, and 2.) From line ($*$) and ($**$), the pre-state to the application expression is equal to the pre-state to both e_1 and e_2. Hence, in terms of the example program's indices,

$$X_1^+ = X_0^+ \qquad\qquad X_2^+ = X_0^+.$$

Now, look at line ($*$) and ($***$). Because expression e' is determined by the output of $E(e_1, s)$, the index of expression e' is determined by the post-state X_1^- of expression e_1. Hence the unknowns for e' are $X_{X_1^-}^+$ and $X_{X_1}^-$. From line ($***$), the pre-state $X_{X_1^-}^+$ to expression e' is "$s[v/x]$", hence

$$X_{X_1^-}^+ = X_0^+[X_2^-/x]. \qquad\qquad (\dagger)$$

Finally, the post-state of the application is equal to the post-state of e'(function body):

$$X_0^- = X_{X_1^-}^-.$$

The equations for the other sub-expressions are similarly derived.

Now, when Zoo solves the established flow equations, because the equations are higher-order, those algorithms that have been developed in the conventional data-flow analysis cannot be used. (The higher-orderedness means that the flow graph (dependences between unknowns) is not fixed *a priori*. For example, see (†).) Hence, different methods[Jør93, CHY95, Bou93] that do not assume a fixed flow graph have to be used in this case.

3 Model Checking the Solutions

Making sense out of the solution of the data-flow equations are done by model-checking. The finite model of the input program is a graph whose nodes are associated with abstract state values. The set of nodes are the pre- and post-states (X_i^+s and X_i^-s) of each expression i[3]. Edge $X_i^\alpha \to X_i^\beta$ exists whenever X_i^α flows into (or, contribute to) X_i^β. When the equations are first-order, the edges are fixed beforehand. Otherwise, they are generated while we solve the equations because the index i is also unknown in the equations. In any case, after solving the equations, we achieve a finite model; the number of nodes (X_i^+s and X_i^-s) are finite and their associated state values are finitely represented (elements of finite lattices).

In Zoo, the query language for the analysis result (the model) is the computational tree logic (CTL)[CGP99, BCM$^+$92, CE81]. That is, we express a program property of our interest in a CTL path formula. For example, at a particular program point, suppose we are interested in:

- Always from this expression, does variable `x` remain positive? This property in CTL is $AG(s(x) =$ "positive"). ("AG" means "for All paths from the current node, Globally every node satisfy the followings.")

[3]In Zoo, the indexing granularity can be larger than expressions, as found useful in balancing the analysis cost-effectiveness[YR01, YR97].

- Sometimes from this expression, can variable `x` is positive? This property in CTL is $EF(s(x) =$ "positive"). ("EF" means "for an Existing path from the current node, some node in the Future satisfy the followings.")
- Always from this expression, does variable `x` remain positive until `y` is negative? Corresponding CTL formula is $AU(s(x) =$ "positive", $s(y) =$ "negative"). (For the modality operators, the reader may want to see [CGP99].)

The checking of such CTL path formula is done by the existing CTL model-checking algorithm.

We have defined a specification language called Rabbit that supports our methodology. For complete syntax and well-formedness rules of the Rabbit, please visit `http://ropas.kaist.ac.kr/zoo/`.

References

[AH95] Alex Aiken and Nevin Heintze. Constraint-based program analysis. Tutorial Notes of the ACM Symposium on Principles of Programming Languages, January 1995.

[BCM+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[Bou93] François Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Lecture Notes in Computer Science*, volume 735 of *Proceedings of Formal Methods in Programming and Their Applications*, pages 128–141. Springer-Verlag, 1993.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic Computation*, 2(4):511–547, 1992. Also as a tech report: Ecole Polytechnique, no. LIX/RR/92/10.

[CC95] Patrick Cousot and Radhia Cousot. Compositional and inductive semantic definitions in fixpoint, equational, constraint, closure-condition, rule-based and game-theoretic form. In *Lecture Notes in Computer Science*, volume 939, pages 293–308. Springer-Verlag, proceedings of the 7th international conference on computer-aided verification edition, 1995.

[CE81] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. *Lecture Notes in Computer Science*, 131:52–71, 1981.

[CGP99] E. M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. The MIT Press, 1999.

[CHY95] Liling Chen, Luddy Harrison, and Kwangkeun Yi. Efficient computation of fixpoints that arise in complex program analysis. *Journal of Programming Languages*, 3(1):31–68, 1995.

[DGS94] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Reducing the cost of data flow analysis by congruence partitioning. In *International Confererence on Compiler Construction*, April 1994.

[GW76] Susan L. Graham and Mark Wegman. A fast and usually linear algorithm for global flow analysis. *Journal of the ACM*, 23(1):172–202, 1976. [Ros80] says rapid monoids are similar to fast monoid in this paper.

[Hec77] Matthew S. Hecht. *Flow Analysis of Comptuer Programs.* Elsevier North-Holland, Inc., 1977.

[Jør93] N. Jørgensen. Chaotic fixpoint iteration guided by dynamic dependency. In *Lecture Notes in Computer Science*, volume 724, pages 27–44. Springer-Verlag, proceedings of the third international workshop on static analysis edition, September 1993.

[KU76] John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithm. *Journal of the ACM*, 23(1):158–171, 1976.

[KU77] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977.

[MR90] Thomas J. Marlowe and Barbara G. Ryder. An efficient hybrid algorithm for incremental data flow analysis. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 184–196, 1990.

[Rey72] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference*, pages 717–740, August 1972.

[Rey98a] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, December 1998. Reprint of [Rey72].

[Rey98b] John C. Reynolds. Definitional interpreters revisited. *Higher-Order and Symbolic Computation*, 11(4):355–361, December 1998.

[Ros80] Barry K. Rosen. Monoids for rapid data flow analysis. *SIAM Journal on Computing*, 9(1), February 1980.

[RP86] Barbara G. Ryder and Marvin C. Paull. Elimination algorithms for data flow analysis. *ACM Computing Surveys*, 18(3):277–316, September 1986.

[SS98] David Schmidt and Bernhard Steffen. Program analysis as model checking of abstract interpretation. volume 1503. Springer-Verlag, proceedings of the 5th static analysis symposium edition, 1998.

[Tar76] Robert Endre Tarjan. Iterative algorithms for global flow analysis. Technical Report Stan-cs-76-547, Stanford University, March 76.

[TJ92] Jean-Piere Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, July 1992.

[TT93] Mads Tofte and Jean-Pierre Talpin. A theory of stack allocation in polymorphically typed languages. Technical Report Technical Report 93/15, Department of Computer Science, Copenhagen University, July 1993.

[YH93] Kwangkeun Yi and Williams Ludwell Harrison III. Automatic generation and management of interprocedural program analyses. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 246–259, January 1993.

[YR97] Kwangkeun Yi and Sukyoung Ryu. Towards a cost-effective estimation of uncaught exceptions in SML programs. In *Proceedings of the 4th International Static Analysis Symposium*, volume 1302 of *Lecture Notes in Computer Science*, pages 98–113. Springer-Verlag, 1997.

[YR01] Kwangkeun Yi and Sukyoung Ryu. A cost-effective estimation of uncaught exceptions in Standard ML programs. *Theoretical Computer Science*, 273(1), 2001. Extended version of [YR97] (to appear).

A About System Zoo

The Zoo system has two aims. We first aim it for being used in our LET project (`ropas.kaist.ac.kr`), in developing our networked ML programming system (named "nML"). The second aim is to transfer the program analysis technology to the industry, hence the generated analyzer's practicality (support for separate analysis and generated analyzer's practical speeds) and friendly user-interface are considered essential. Support for constraint-based analysis[AH95] and type-based analysis[TJ92, TT93] are also planned.

As of the specification language *per se*, we have developed its typing system such that well-formed specification guarantees to transform into typeful ML programs. We are in the middle of developing a static check for monotonicity, which is essential to guarantee the analysis termination. A specification in Zoo consists of domains (lattice and set definitions), recursive abstract interpreter functions, and CTL formula. From the specification Zoo generates three executable programs in ML and C, which compile the input program into the set of equations, solve the equations (constructing finite models), and check the CTL formula.