Relational Abstract Interpretation of Higher Order Functional Programs (extended abstract)

Patrick COUSOT & Radhia COUSOT

LIX, URA CNRS 1439
École Polytechnique, F-91128 Palaiseau Cedex, France
email: `cousot@polytechnique.fr`

1 Approximating functions by functions

Most applications of the abstract interpretation framework[2] have been for analyzing functional programs use functions on abstract values to approximate functions, thus assuming that functions may be called at all arguments. When the abstract domain is finite, this approach can easily be generalized to higher order functional languages as shown for example by [1]. In practice this leads to combinatorial explosion problems as observed, for example, in strictness analysis of higher order functional languages.

2 Minimal Function Graphs

However it is often only necessary to analyze a functional program for inputs satisfying a given specification so that functions are called only for some the possible abstract values of their arguments. Therefore we proposed [3] to solve the fixpoint system of equations $\phi_i(X) = F_i(\phi)(X), i \in [1, n]$ (corresponding to a first order functional program) on the necessary arguments only, an idea later popularized by Jones and Mycroft under the name of *minimal function graphs*. Unfortunately, this minimal function graphs approach has several defects :

1. Functions are approximated by functions (or function graphs which is equivalent), which is not general and/or simple enough, as shown by type checking where a function is more simply approximated by the tuple of types of its arguments and result;
2. The minimal function graph semantics first consists in collecting the set of all abstract arguments X on which each user defined function

ϕ_i is to be or has been calculated, then in deriving the corresponding value $F_i(\phi)(X)$ as well as any further arguments of ϕ_j which might be needed in order to evaluate $F_i(\phi)$ on X, in joining them together to yield an updated set of needed arguments and in iterating so on until stabilization. This algorithm essentially corresponds to a forward analysis, i.e. in the direction of the flow of control. It is not well suited at all for backward analyses;

3. Since the analysis of a function cannot be done without knowing the values of its arguments, it is not well adapted for separate analyses of (say non-mutually recursive) functions (as the type checker is able to do in ML). Hence it is not easily usable in separate compilations.

3 Relational Abstract Interpretation of First Order Functional Programs

In [3] we suggested another method with consists in approximating functions (or procedures in the case of imperative programs) by relations. These relations can be further approximated by linear inequalities between values of variables [4]. Let us illustrate this method using a PASCAL example taken from [6]:

```
procedure Hanoi (n : integer; var a, b, c : integer;
                              var Ta, Tb, Tc : Tower);
begin
  { n = n0 ∧ a = a0 ∧ b = b0 ∧ c = c0 }
  if n = 1 then begin
    b := b + 1; Tb[b] := Ta[a]; Ta[a] := 0; a := a - 1;
    { n = n0 = 1 ∧ a = a0 - 1 ∧ b = b0 + 1 ∧ c = c0 }
  end else begin
    { n = n0 ∧ a = a0 ∧ b = b0 ∧ c = c0 }
    Hanoi(n - 1, a, c, b, Ta, Tc, Tb);
    { n = n0 > 1 ∧ a = a0 - n + 1 ∧ b = b0 ∧ c = c0 + n - 1 }
    b := b + 1; Tb[b] := Ta[a];  Ta[a] := 0; a := a - 1;
    { n = n0 > 1 ∧ a = a0 - n ∧ b = b0 + 1 ∧ c = c0 + n - 1 }
    Hanoi(n - 1, c, b, a, Tc, Tb, Ta);
    { n = n0 > 1 ∧ a = a0 - n ∧ b = b0 + n ∧ c = c0 }
  end;
  { n = n0 ≥ 1 ∧ a = a0 - n0 ∧ b = b0 + n0 ∧ c = c0 }
end;
```

The result of analyzing this procedure, which is given above between brackets $\{\ldots\}$ is independent of the values of the actual parameters provided in calls. This is obtained by giving formal names n_0, a_0, b_0 and c_0 to the values of the actual parameters corresponding to the initial values of the formal parameters `n`, `a`, `b` and `c` (array parameters `Ta`, `Tb` and `Tc` are simply ignored) and by establishing a relation with the final value of these formal parameters. The result is a precise description of the effect of the procedure in the form of a relation between initial and final values of its parameters:

$$\phi(n_0, a_0, b_0, c_0, n, a, b, c) = (n = n_0 \geq 1 \wedge a = a_0 - n_0 \wedge b = b_0 + n_0 \wedge c = c_0)$$

Observe that it is automatically shown that $n_0 \geq 1$ is necessary (for termination).

In a function call, n_0, a_0, b_0 and c_0 are set equal to the values of the actual parameters in ϕ and eliminated by existential quantification. For example:

```
a := n; b := 0; c := 0;
{ n = a ∧ b = 0 ∧ c = 0 }
Hanoi(n, a, b, c, Ta, Tb, Tc);
{ ∃n0, a0, b0, c0 : n0 = a0 ∧ b0 = 0 ∧ c0 = 0 ∧ n = n0 ≥ 1 ∧ a = a0 − n0
∧ b = b0 + n0 ∧ c = c0 }
```

This last post-condition can be simplified by projection as :

$$\{\ a = 0 \ \wedge\ n = b \geq 1 \ \wedge\ c = 0\ \}$$

In recursive calls, successive approximations of the relation ϕ must be used, starting from the empty one. A widening (followed by a narrowing) [2] can be used to ensure convergence.

This method can be extended to a functional language using a relation between the formal values of the parameters of a function and its result.

4 Relational Abstract Interpretation of Higher Order Functional Programs

The object of this paper is to extend this technique to higher order functional programs (without calling on higher order relations). The advantages of doing so are the following:

1. Functions are approximated by relations, which can be represented formally as predicates on formal variables and further approximated to get compact and easily computer-representable abstract values as illustrated by [7], [6] and [5];
2. Widening and narrowing operators of [2] are used to enforce convergence (so that the lattice need not satisfy the ascending chain condition as in [8]);

3. Forward and backward analyses are handled in the same way;
4. Groups of mutually recursive functions can be analyzed separately.

References

[1] Burn, G.L., Hankin, C.L. & Abramsky, S. "The Theory and Practice of Strictness Analysis for Higher Order Functions", *Science of Computer Programming*, vol. 7, 1986, pp. 249–278.

[2] Cousot, P. & Cousot, R. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". Conference Record of the 4$^{\text{th}}$ ACM Symposium on Principles of Programming Languages, Los Angeles, California, U.S.A., 1977, pp. 238–252.

[3] Cousot, P. & Cousot, R. "Static Determination of Dynamic Properties of Recursive Procedures". *Formal Description of Programming Concepts*, E.J. Neuhold (ed.), North-Holland, Amsterdam, 1977, pp. 237–277.

[4] Cousot, P. & Halbwachs, N. "Automatic discovery of linear restraints among variables of a program". Conference Record of the 5$^{\text{th}}$ ACM Symposium on Principles of Programming Languages, Tucson, Arizona, U.S.A., 1978, pp. 84–97.

[5] Granger, P. "Static Analysis of Linear Congruence Equalities among Variables of a Program", Proceedings of the Fourth International Joint Conference on the Theory and Practice of Software Development (TAPSOFT'91), Brighton, UK, April 1991, Vol. 1 (CAAP'91), LNCS 493, Springer-Verlag, pp. 169-192.

[6] Halbwachs, N. "Détermination automatique de relations linéaires vérifiées par les variables d'un programme", Thèse de 3$^{\text{ème}}$ cycle, Université Scientifique et Médicale de Grenoble, 12 mars 1979.

[7] Karr, M. "Affine Relationships among Variables of a Program", *Acta Informatica*, vol. 6, 1976, pp. 188–206.

[8] Jones, N. D. & Mycroft, A. "Data flow analysis of applicative programs using minimal function graphs". Conference Record of the 13$^{\text{th}}$ ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, U.S.A., 1986, pp. 286–306.