Abstract Interpretation

Sorin Lerner

June 4, 2001

1 Introduction

The work surveyed in class so far was not concerned with proving correctness of dataflow analyses. We saw algorithms for computing properties of a program, but it was not shown that these properties correctly characterized the program. Although it would be possible to reason about the correctness of dataflow analyses one at a time, a unified framework would make the proofs easier to understand and to develop. Abstract interpretation is exactly such a framework: it provides a unified theory of fixpoint approximations which (amongst other things) can be used to formalize the notion of correctness of a dataflow analysis with respect to the semantics of a program. This is achieved by relating the solution of a dataflow analysis with the behaviors of a program, and guaranteeing that the dataflow solution over-estimates the possible behaviors of the program.

2 Program Semantics and Fixpoints

The semantics of a programming language specifies the semantics of any program written in that language. The semantics of a program provides a mathematical model that specifies all possible behaviors of the program [1]. Cousot and Cousot argue that the semantics of a program can be expressed as the least fixpoint of a function F_c (a fixpoint being a solution to the equation $X = F_c(X)$). F_c is called the concrete interpretation function, and it specifies the concrete semantics of a program.

Example. Here is a simple example showing how a fixpoint can be used to encode the behaviors of a program. Assuming a control flow graph representation, the state of execution of a program is a pair (a, e), where a is an edge in the control flow graph, and e is an environment that maps variables to values. Let $States$ be the set of all possible states of a program, and ι be the initial state of the program (the state in which the program starts). The concrete interpretation function $F_c : 2^{States} \rightarrow 2^{States}$ can then be defined as follows: $F_c(X)$ returns a set of states which includes ι as well as all states that can be reached from the states in X using one operation in the control flow graph. The least fixpoint of F_c is the set of all possible states that the program can reach when started in the initial state ι. If the program runs forever, the least fixpoint of F_c still exists but will be infinite. ■

Solving for the least fixpoint in the concrete semantics of a program is generally undecidable because the program may not terminate and the set of possible inputs to the program may be infinite. Even if the program *does* terminate and the set of inputs *is* finite, finding the least fixpoint of F_c is tantamount to running the program, which may be too expensive for static analysis. The key idea then is to approximate the least fixpoint of F_c by finding the least fixpoint of an appropriate abstract interpretation function

F_a. The abstract interpretation function operates over an abstract domain which only keeps track of a particular property of interest. As a result the abstract domain is not as precise as the concrete domain, but it is also "smaller" which usually means that the least fixpoint of F_a is computable. Once we have the least fixpoint of F_a in the abstract domain, it is possible to map this value back into the concrete domain, resulting in an approximation to the fixpoint of F_c. An approximation in this context means that the approximate fixpoint of F_c over-estimates the real fixpoint: we will conclude that the program can have more behaviors than it actually has.

Intuitively, finding the fixpoint of F_c amounts to running the program in the concrete domain, whereas finding the fixpoint of F_a amounts to running the program in the abstract domain, for instance running a dataflow analysis algorithm.

Example. Suppose we are interested in the signs of integers flowing in a program, but not the exact values. In the concrete domain we store for each program point a map from variables to sets of integers. In the abstract domain, we store for each program point a map from variables to an element from the abstract domain $\{+, -, \pm\}$. $+$ stands for positive numbers, $-$ for negative numbers and $\pm$ for positive or negative numbers. In the concrete domain, $\times$ operates over integers whereas in the abstract domain it operates over the abstract values $\{+, -, \pm\}$. For example -15×17 evaluates to -180 in the concrete domain, which maps to "$-$" $\times$ "$+$" evaluating to "$-$" in the abstract domain. As another example, $-15 + 17$ evaluates to 2 in the concrete domain, whereas "$-$" + "$+$" evaluates to "$\pm$" in the abstract domain. ■

The above example shows the two ways in which the abstract domain loses precision. First, there is a loss of precision due to the abstraction itself: even though the result of -15×17 is -180, the most precise answer the abstract domain can give is that the result is negative. Second, there is a loss of precision because the operators in the abstract domain reach fixpoint faster, and therefore lose precision more quickly: even though the result of $-15 + 17$ is positive, the best we can do in the abstract domain is conclude that the result is either positive or negative.

3 Abstract Interpretation

3.1 Definition

This section formalizes the intuitive ideas presented in the previous section. We have already seen how both the semantics of a program and the dataflow analysis algorithm can be seen as fixpoint computations. We formalize this by introducing the general notion of an abstract interpretation: an abstract interpretation is a tuple $I = \langle D, \sqcup, \sqcap, \sqsubseteq, \top, \bot, F \rangle$, where $\langle D, \sqcup, \sqcap, \sqsubseteq, \top, \bot \rangle$ is a complete lattice, and $F : D \to D$ is the interpretation function (see appendix A for the definition of a complete lattice). The solution of an abstract interpretation $I = \langle D, \sqcup, \sqcap, \sqsubseteq, \top, \bot, F \rangle$ is the least fixpoint of F [1]. Assuming F is continuous, then the least fixpoint of F is $\bigsqcup_{i=0}^{\infty} F^i(\bot)$ (see appendix A).

Intuitively, $\bot$ represents no information about the program, and by applying F we gather more and more information until we reach fixpoint. Elements higher in the lattice approximate those lower in the lattice because higher elements encode more behaviors. Therefore, $\bot$ (assume nothing happens) is the most optimistic information, and $\top$ (assume anything can happen) is the most conservative information [2].

[1] If you have read [2], you'll notice that Cousot and Cousot define $I = \langle \text{A-cont}, \sqcup, \sqsubseteq, \top, \bot, Int \rangle$, and the solution of I as the fixpoint of $\widetilde{Int}$, an extension of Int. My definition differs slightly in that the interpretation function F_I is the actual function of which we are finding the fixpoint.

[2] Note that the literature on dataflow analysis uses the opposite convention: $\bot$ as conservative and $\top$ as optimistic.

3.2 Consistent Abstract Interpretation

In section 2, we talked about two abstract interpretations: the concrete semantics $I_c = \langle D_c, \sqcup_c, \sqcap_c, \sqsubseteq_c, \top_c, \bot_c, F_c \rangle$, and an abstract version of the concrete semantics $I_a = \langle D_a, \sqcup_a, \sqcap_a \sqsubseteq_a, \top_a, \bot_a, F_a \rangle$. As we saw I_c is more precise than I_a but its fixpoint may not be computable. The goal is to approximate the fixpoint of I_c by finding a fixpoint of I_a. This can be formalized by relating the two fixpoints (and more generally relating concrete and abstract values) using an *abstraction function* $\alpha : D_c \rightarrow D_a$ and a *concretization function* $\gamma : D_a \rightarrow D_c$. The abstraction function returns the abstract property that an exact behavior has. The concretization function returns the concrete values that have a given abstract property.

The relation that we require between the fixpoint of an "abstract" interpretation I_a and a "concrete" interpretation I_c is shown below:

$$\alpha \left(\bigsqcup_{i=0}^{\infty} F_c^i (\bot_c) \right) \sqsubseteq_a \bigsqcup_{i=0}^{\infty} F_a^i (\bot_a) \tag{1}$$

$$\bigsqcup_{i=0}^{\infty} F_c^i (\bot_c) \sqsubseteq_c \gamma \left(\bigsqcup_{i=0}^{\infty} F_a^i (\bot_a) \right) \tag{2}$$

These properties formalize the intuitive idea that we want the dataflow analysis to over-estimate the actual behaviors of the program. More precisely, property (1) says that the analysis has done enough work to over-approximate the property we are interested in, whereas property (2) says that the analysis has done enough work to over-approximate the behavior of the program.

Properties (1) and (2) are global in that they hold at fixpoint. Because these global properties are hard to prove directly, Cousot and Cousot introduce the following local consistency properties, which are sufficient to prove (1) and (2):

$$\alpha \text{ and } \gamma \text{ are continuous} \tag{3}$$

$$\forall x \in D_c . \alpha(F_c(x)) \sqsubseteq_a F_a(\alpha(x)) \tag{4}$$

$$\forall x \in D_a . F_c(\gamma(x)) \sqsubseteq_c \gamma(F_a(x)) \tag{5}$$

Condition (3) requires α and γ to be continuous and we will shortly see where this is used. Property (4) essentially says that mapping over to the abstract domain and taking one step there will bring you higher (recall that higher means over-estimating) in the abstract lattice than taking one step in the concrete domain and then mapping over to the abstract domain. Property (5) has a similar intuitive interpretation.

Properties (4) and (5) are in fact local versions of (1) and (2). For example, assuming property (4) it is possible to show by induction that

$$\forall i \geq 0 . \alpha(F_c^i(\bot_c)) \sqsubseteq_a F_a^i(\bot_a) \tag{6}$$

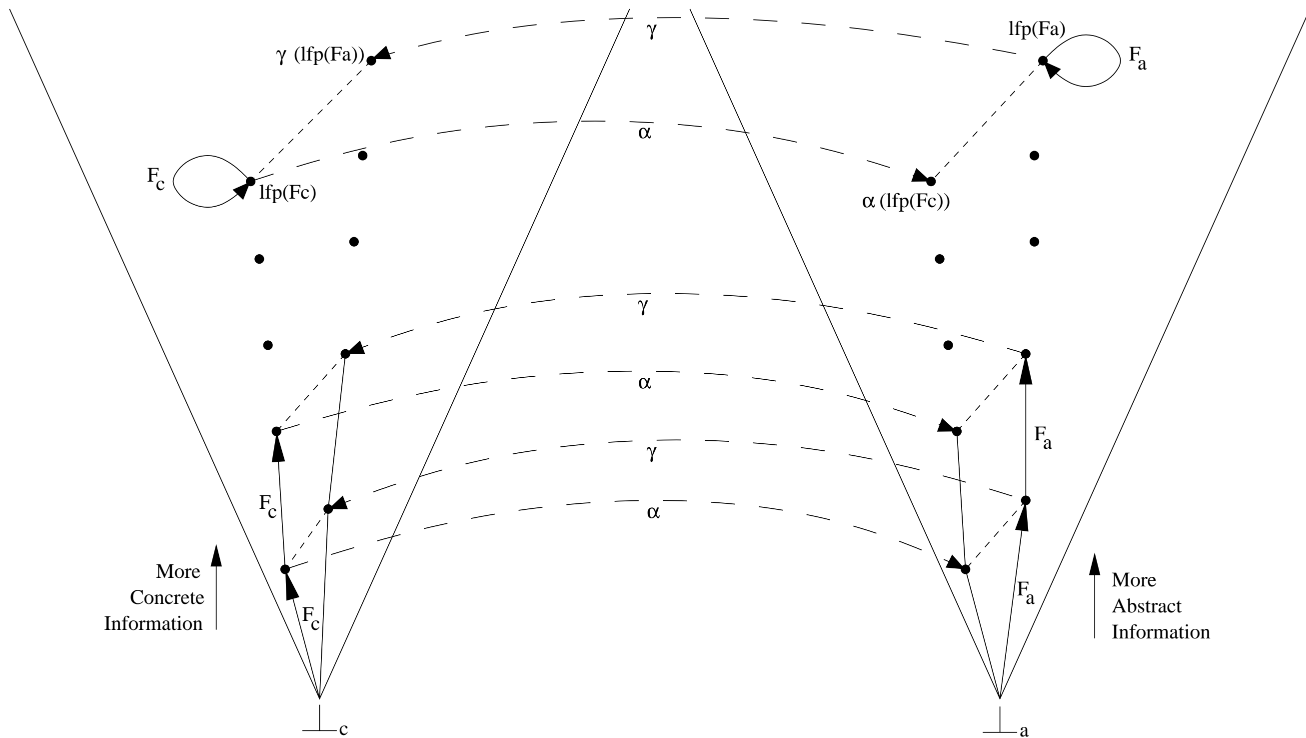

Figure 1: Connection between two abstract interpretations I_c and I_a. If a straight line (dashed, solid, or an arrow) connects two points, then the two points are ordered in the lattice: the point higher up in the drawing is the larger one in the lattice.

Property (6) and the continuity of α then imply property (1). The continuity of α is important here because it justifies the step from property (6), which talks about finite i, to property, (1) which talks about infinite chains.

Figure 1 helps visualize the connection between the abstract and the concrete domain. It shows the successive steps in the fixpoint computation of F_c and F_a, and how α and γ relate the domain elements computed at each step. For example, notice that there is a straight dashed line between the points that represent $\alpha(F_c(F_c(\bot_c)))$ and $F_a(F_a(\bot_a))$. This indicates that after two steps we have $\alpha(F_c(F_c(\bot_c))) \sqsubseteq_a F_a(F_a(\bot_a))$, which is an instantiation of property (6). Similarly, notice how there is a straight dashed line between $\alpha(lfp(F_c))$ and $lfp(F_a)$ (where lfp is the least fixpoint operator, returning the least fixpoint of a function – see appendix A for details), indicating that $\alpha(lfp(F_c)) \sqsubseteq_a lfp(F_a)$ (property (1)). There is also a straight dashed line to indicate that $lfp(F_c) \sqsubseteq_c \gamma(lfp(F_a))$ (property (2)).

3.3 Continuity of α and γ

In [2] Cousot and Cousot do not directly require α and γ to be continuous. Instead they require that the following stronger properties hold, which in turn imply the continuity of α and γ:

$$\forall x \in D_c . x \sqsubseteq \gamma(\alpha(x)) \tag{7}$$

$$\forall x \in D_a . x = \alpha(\gamma(x)) \tag{8}$$

Equation (8) says that concretization introduces no loss of precision: the concretization of an abstract property has exactly that property. Equation (7) says that abstraction may lead to loss of precision: if a program's behaviors are abstracted and then concretized, we might end up with an over-estimation of the original behaviors (that is, a larger set of behaviors).

In addition to guaranteeing the continuity of α and γ, property (8) also guarantees that α and γ are non-trivial. If the only requirement on α and γ were continuity, then one could define $\alpha(x) = \bot_a$ for all x and $\gamma(x) = \top_c$ for all x, which would trivially satisfy (4) and (5). Property (8) prevents us from doing this by requiring that α and γ encode in a non-trivial way the abstract property of interest.

3.4 Summary of Requirements

In summary, here are the steps required to use abstract interpretation to show that an analysis algorithm is correct with respect to the semantics of a program:

- Define the concrete domain and the abstract domain.
- Define F_c (the semantics of programs) and F_a (the analysis algorithm), and show that they are continuous.
- Define α and γ in a way that encodes the abstract property of interest, and show that they are continuous.
- Prove that local consistency properties (4) and (5) hold for F_c and F_a.

Once these requirements have been fulfilled, properties (1) and (2) are guaranteed to hold.

4 Further Reading

In this paper summary we have only looked at one application of abstract interpretation, namely its use as a correctness criterion for static analyses. Abstract interpretation is however much more general than this and has been applied in many settings.

For example, Chapter 3 of [5] shows how to use abstract interpretation to reason about Binding Time Analyses. The framework is different from the one presented in this paper summary because it does not use γ. Indeed, if we are only interested in showing (1), then it is sufficient to define α in such a way that it encodes the property of interest, and then show (4). This can be useful if γ is hard to define.

As another example, abstract interpretation has been used in model checking. The model checking community has developed a notion of abstraction [8] which is similar to abstract interpretation. The main difference is that the abstraction framework from model checking does not require an ordering on the concrete and abstract values. There has however been work on model checking that does impose such an ordering, and uses the general framework of abstract interpretation (eg: [4], [7]). As another example, Cousot and Cousot have shown how to study temporal calculi and logics using abstract interpretation [3].

Finally, the extended electronic version of [1], which can be found at `http://www.di.ens.fr/~cousot/COUSOTpapers/LNCS2000-01.shtml`, has a large bibliography on the topics of abstract interpretation, program analysis and formal methods. The interested reader can use this bibliography as a good starting point for further material.

A Partially Ordered Sets and Lattices

This section presents some background on partially ordered sets and lattices. It is meant to cover only those topics which are required to understand this paper summary. More details can be found in Appendix A of [9] or in [6].

Partially ordered sets. A relation R over a set D is a subset of $D \times D$. A relation is a *partial order* iff it is reflexive, transitive and anti-symmetric. A *partially ordered set*, or *poset* $(D, \sqsubseteq)$ is a set D equipped with a partial order $\sqsubseteq$ over D.

Upper and lower bounds. If $Y \subseteq D$ is a subset of a poset $(D, \sqsubseteq)$ then:

- d is an *upper bound* of Y iff $d \in D$ and $\forall d' \in Y. d' \sqsubseteq d$.
- d is the *least upper bound* of Y, denoted $\bigsqcup Y$, iff d is an upper bound of Y and $d \sqsubseteq d'$ holds for any upper bound d' of Y.
- d is a *lower bound* of Y iff $d \in D$ and $\forall d' \in Y. d \sqsubseteq d'$.
- d is the *greatest lower bound* of Y, denoted $\bigsqcap Y$, iff d is a lower bound of Y and $d' \sqsubseteq d$ holds for any lower bound d' of Y.

Chains. A subset $Y \subseteq D$ of a poset $(D, \sqsubseteq)$ is a chain iff $\forall d, d' \in Y. (d \sqsubseteq d' \vee d' \sqsubseteq d)$

Complete Lattices. A complete lattice is a partially ordered set where each subset has a least upper bound and a greatest lower bound. Formally, a complete lattice is a tuple $L = \langle D, \sqcup, \sqcap, \sqsubseteq, \top, \bot \rangle$, where:

- $(D, \sqsubseteq)$ is a poset.
- $\bigsqcup$ is the least upper bound operator and $\bigsqcup Y$ exists for any $Y \subseteq D$.
- $\bigsqcap$ is the greatest lower bound operator and $\bigsqcap Y$ exists for any $Y \subseteq D$.
- $\top$ is the greatest element of D and $\bot$ is the least element in D.

Monotonicity. A function f from one complete lattice $\langle D, \sqcup, \sqcap, \sqsubseteq, \top, \bot \rangle$ to another $\langle D', \sqcup', \sqcap', \sqsubseteq', \top', \bot' \rangle$ is monotone iff $x \sqsubseteq y \rightarrow f(x) \sqsubseteq' f(y)$.

Continuity. A function f from one complete lattice $\langle D, \sqcup, \sqcap, \sqsubseteq, \top, \bot \rangle$ to another $\langle D', \sqcup', \sqcap', \sqsubseteq', \top', \bot' \rangle$ is continuous iff for any chain Y it is the case that $\bigsqcup' \{f(d) | d \in Y\} = f(\bigsqcup Y)$. If f is continuous, then f is monotone.

Example. Here is an example of a function that is not continuous. Let $D = \{0, 1, 2, ...\} \cup \infty$ and $L = \langle D, \sqcup, \sqcap, \leq, \infty, 0 \rangle$; $D' = \{0, 1\}$ and $L' = \langle D', \sqcup', \sqcap', \leq, 1, 0 \rangle$. Let $f : D \rightarrow D'$ be defined as follows:

$$f(x) = \begin{cases} 0 & \text{if } x \text{ is finite} \\ 1 & \text{if } x = \infty \end{cases}$$

Let Y be the set of finite integers. Y is a chain because for any two integers a and b it is the case that $a \leq b \vee b \leq a$. We have $\bigsqcup Y = \infty$, and so $f(\bigsqcup Y) = 1$. However, because $f(x) = 0$ whenever x is finite we have $\{f(d) | d \in Y\} = \{0\}$, and so $\bigsqcup' \{f(d) | d \in Y\} = 0$. Therefore f is not continuous. It is however easy to see that f is monotone. ■

Fixpoints. A *fixpoint* of a function $f : D \rightarrow D$ over a complete lattice $\langle D, \sqcup, \sqcap, \sqsubseteq, \top, \bot \rangle$ is an element $d \in D$ such that $f(d) = d$. The least fixpoint of f, denoted $lfp(f)$, is a fixpoint d such that $d \sqsubseteq d'$ holds for any fixpoint d' of f.

If f is continuous, then f has a least fixpoint, and $lfp(f) = \bigsqcup_{i=0}^{\infty} f^i(\bot)$, where $f^0 = \lambda x.x$ and $f^k = f \circ f^{k-1}$ for $k > 0$.

References

[1] P. Cousot. Abstract interpretation based formal methods and future challenges. In *Informatics, 10 Years Back - 10 Years Ahead*, Lecture Notes in Computer Science 2000. R. Wilhelm (Ed.), 2001.

[2] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for statis analysis of programs by construction or apporximation of fixpoints. In *Proceedings of the Fourth ACM Symposium on Principles of Programming Languages*, Los Angeles CA, January 1977.

[3] P. Cousot and R. Cousot. Temporal abstraction. In *Proceedings of the 27-th ACM Symposium on Principles of Programming Languages*, Boston MA, January 2000.

[4] Dennis Dams, Rob Gerth, and Orna Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, March 1997.

[5] Manuvir Das. *Partial Evaluation Using Dependence Graphs.* PhD thesis, University of Wisconsin - Madison, Feb 1998.

[6] B.A. Davey and H. A. Prestley. *Introduction to Lattices and Order*. Cambridge Mathematical Textbooks. Cambridge University Press, 1990.

[7] F. Huch. Verification of erlang programs using abstract interpreation and model checking. In *ACM SIGPLAN Notices*, pages 34(9):261–272, September 1999. Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 99).

[8] E. M. Clarke Jr., O. Grumberg, and D. E. Long. Model checking and abstraction. In *Proceedings of the Nineteenth ACM Symposium on Principles of Programming Languages*, Albequerque New Mexico, January 1992.

[9] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.