Analysis of Programs

Program analysis offers static (i.e. compile-time) techniques for predicting safe and computable approximations to the set of values or behaviours arising dynamically (i.e. at runtime) when executing a program upon a computer. A main application is to allow compilers to generate more efficient code by avoiding to generate code to handle errors that can be shown not to arise, and to generate more specialised and faster code by taking advantage of the availability of data in registers and caches. Among the newer applications is the validation of software (possibly developed by subcontractors) to reduce the likelihood of malicious behaviour. To illustrate the breadth of the approaches we shall explain (i) flow-based analysis, (ii) inference-based analysis, (iii) semantics-based analysis, (iv) abstract interpretation, and (v) fixed point approximation techniques. These approaches borrow mathematical techniques from lattice theory and logical systems.

Flow-based approaches to program analysis have mainly been developed for imperative, functional, and object-oriented programs. For a given program $p$ the task is to find a flow function $F \in \text{Inf}^{\text{Pts}}$ that maps the program points in $p$ to approximate descriptions of the data that may reach those points. This may be formalised by a relation $\models p : F$ defined by a number of clauses depending on the syntax of the program $p$. As a very simple example consider the program $p = p_1; p_2$ that takes an input, executes $p_1$ on it, then executes $p_2$ on the result obtained from $p_1$, and finally produces the result from $p_2$ as the overall result. The defining clause for this program may be written:

$$\models (p_1; p_2) : F \text{ if and only if}$$

$$\models p_1 : F \land$$

$$\models p_2 : F \land$$

$$F(\text{end}(p_1)) \subseteq F(\text{end}(p_2))$$

In the presence of recursive programs the predicate cannot be defined by structural induction on $p$ and instead one appeals to co-induction; this merely amounts to taking the greatest fixed point of a function mapping sets of pairs $(p, F)$ into sets of pairs $(p, F)$ as described by the right hand sides of the clauses.

It is essential that the analysis information is semantically correct. Given a semantics $v_1 \rightarrow_p v_2$ for describing when the program $p$ maps the input $v_1$ to the output $v_2$, this amounts in its simplest form to establishing a subject reduction result:

$$\models p : F \land v_1 \in F(\text{begin}(p)) \land v_1 \rightarrow_p v_2$$

$$\Downarrow$$

$$v_2 \in F(\text{end}(p))$$

Finally realisation of the amount of data to finding the minimal $F$ (wrt. the point-wise partial order on $\text{Inf}^{\text{Pts}}$) for which $\models p : F$; this is related to the lattice theoretic notion of $\{F \models p : F\}$ being a Moore family. Often this is done by obtaining a more implementation-oriented specification $\vdash p : C$ for generating a set of contraints $C$, deriving a function $C$ mapping flow functions to flow functions, and ensuring that $\overline{C}(F) = F$ if and only if $\models p : F$.

Inference-based approaches to program analysis have mainly been developed for functional, imperative, and concurrent programs. For a given program $p$ the task is to find an “annotated type” $q_1 \xrightarrow{\sigma_1} q_2$ describing the overall effect of executing $p$; if $q_1$ describes the input data, then $q_2$ describes the output data, and properties of the dynamic executions (e.g. whether or not errors can occur) is described by $\sigma$. This may be formalised by a relation $\vdash p : (q_1 \xrightarrow{\sigma_1} q_2)$ defined by a number of inference rules; some of these are defined compositionally in the structure of the program as in:

$$\vdash p_1 : (q_0 \xrightarrow{\sigma_1} q_1) \quad \vdash p_2 : (q_1 \xrightarrow{\sigma_2} q_2)$$

$$\vdash (p_1; p_2) : q_0 \xrightarrow{\sigma_1 \cup \sigma_2} q_2$$

Here $\sigma_1 \cup \sigma_2$ combines the effects $\sigma_1$ and $\sigma_2$ into an overall effect; in the simplest cases $\sigma_1 \cup \sigma_2$ simply equals $\sigma_1 \cup \sigma_2$ but in more complex cases the causal structure (i.e. which effect comes first) must be retained. Successful analysis of recursive programs demands that invariants be guessed in order to obtain a finite inference tree. (This form of program analysis is closely related to program verification; see “Theoretical Programming” for the paragraph “The logical theory of programming”.)

Semantic correctness necessitates relations $\models v : q$ between values and properties, e.g. $(q = \text{integer}) \Rightarrow (v \in \{\cdots, -1, 0, 1, \cdots\})$, and $\models d \sim_\sigma$ between the effects of the dynamic semantics $(v_1 \rightarrow_p v_2)$ and the analysis,
e.g. \( d = \sigma \). The correctness statement then amounts to a subject reduction result:

\[
\frac{\Gamma \vdash v_1 : q_1 \land (v_1 \rightarrow_P v_2) \land \vdash p : (q_1 \Rightarrow \sigma \Rightarrow q_2)}{\vdash v_2 : q_2 \land (\vdash d \sim \sigma)}
\]

Implementation of the analysis amounts to defining an algorithm \( Q[p] = q \) that is syntactically sound, i.e. \( \vdash p : q \), and syntactically complete, i.e. \( \vdash p : (q_1 \Rightarrow \sigma \Rightarrow q_2) \) implies that \( q_1 \Rightarrow \sigma \Rightarrow q_2 \) is an “instance” of \( q \); the precise definition of “instance” depends on the nature of the properties and may have some complexity to it. Often the algorithm works using constraints somewhat similar to those used in the flow-based approach.

**Semantics-based approaches** to program analysis attempt to define the analysis in much the same way that the semantics itself is defined. These developments have mainly been based on denotational semantics (where \( v_1 \rightarrow_p v_2 \) is written \( S[p](v_1) = v_2 \)) for imperative and functional programs, although work has started on similar developments based on (so-called small-step) structural operational semantics and (so-called big-step) natural semantics. The denotational approach may be formulated as an analysis function \( A[p](l) = l_2 \) where \( l_1 \) is a mathematical entity describing the input to \( p \) and \( l_2 \) is a mathematical entity describing the output from \( p \). The denotational approach calls for \( A[p] \) to be defined compositionally in terms of the syntactic constituents of the program \( p \); an example of this is the functional composition

\[
\]

i.e. \( A[p_1; p_2](l) = A[p_2](A[p_1](l)) \). In case of recursive programs a least fixed point is taken of a suitable functional.

To express the correctness of the analysis we need to define correctness relations \( v \mathrel{R} l \) on the data, e.g. \( v \in l \), and \( f \mathrel{R_{t_1 \rightarrow t_2}} h \) on the computations, e.g. \( \forall v, l : (v \mathrel{R_{t_1 \rightarrow t_2}} l) \Rightarrow (f(v) \mathrel{R_{t_2 \rightarrow t_2}} h(l)) \); the latter definition is usually called a logical relation. Semantic soundness then amounts to establishing

\[
S[p] \mathrel{R_{t_1 \rightarrow t_2}} A[p]
\]

for all programs \( p \) of type \( t_1 \rightarrow t_2 \). Efficient implementation of the analyses often demands that they be presented in a form less dependent on the program syntax, e.g. using graphs or constraints, and then the solution procedure must be shown sound and complete.

**Abstract interpretation** is a general framework for obtaining cheaper and more approximative solutions to program analysis problems. In its most general form it attempts to be independent of the semantic base, but usually it is formulated in either a structural operational or a denotational setting. Given the semantics \( v_1 \rightarrow_p v_2 \) of a program \( p \), the first step is often to define the uncomputable collecting semantics \( s_1 \Rightarrow_p s_2 \). The intention is that

\[
s_1 \Rightarrow_p s_2 \Downarrow s_2 = \{ v_2 \mid v_1 \rightarrow_p v_2 \land v_1 \in s_1 \}
\]

and this may be regarded as a soundness as well as a completeness statement.

To obtain a more approximate and eventually computable analysis the most commonly used approach is to replace the sets of values \( s \in S \) by some more approximate properties \( l \in L \). Both \( S \) and \( L \) are partially ordered in such a way that we generally want the least correct elements and where the greatest element is uninformative. (By the lattice duality principle we could instead have desired the greatest correct element, and part of the literature on data flow analysis uses this vocabulary.) The relation between \( S \) and \( L \) is often formalised by a pair \((\alpha, \gamma)\) of functions:

\[
S \stackrel{\gamma}{\Rightarrow} L \mathrel{\alpha}
\]

where \( \gamma : L \rightarrow S \) is the concretisation function giving the “meaning” of properties in \( L \), and \( \alpha : S \rightarrow L \) is the abstraction function. It is customary to demand that \( S \) and \( L \) are complete lattices. Furthermore it is customary to demand that \((\alpha, \gamma)\) is a Galois connection, i.e. \( \alpha \) and \( \gamma \) are monotone and \( \alpha \circ \gamma \) is extensive (\( \forall s : (\alpha(\gamma(s)) \supseteq s) \)) and \( \alpha \circ \gamma \) is reductive (\( \forall l : (\alpha(\gamma(l)) \supseteq l) \)), or equivalently that \((\alpha, \gamma)\) is an adjunction, i.e. \( \forall s, l : (\alpha(s) \subseteq l \iff s \subseteq \gamma(l)) \). This ensures that a set of values \( s \) always has a best description \( l \) : in fact \( l = \alpha(s) \).

Abstract interpretation then amounts to define an analysis \( l_1 \Rightarrow_p l_2 \) on the properties of \( L \) such that the following correctness statement
Fixed point approximation is essential for program analysis because most of the approaches can be reformulated as finding the least fixed point $lfp(f)$ of a monotone function $f : L \rightarrow L$ upon a complete lattice $L = (L, \sqsubseteq)$ of properties; here each element $l \in L$ often describes a set of values. There are two characterizations of the least fixed point that are of relevance here. A theorem by Tarski ensures that

$$lfp(f) = \sqcap Red(f)$$

where $Red(f) = \{ l \in L | f(l) \sqsubseteq l \}$

satisfies $f(lfp(f)) = lfp(f)$ and hence $lfp(f) \in Red(f)$ is the least fixed point of $f$. Transfinite induction allows to define

$$f^{n+1} = f(f^n)$$

for successor ordinal $\kappa$

$$f^\kappa = \sqcup_{\kappa < \kappa} f^n$$

for limit ordinal $\kappa$

and then the least fixed point is given by

$$lfp(f) = f^{\lambda}$$

for some cardinal $\lambda$; it suffices to let $\lambda$ be the cardinality of $2^L$. In simple cases $L$ has finite height (all strictly increasing sequences are finite) and then $\lambda$ may be taken to be a natural number $n_0$; this then suggests a computable procedure for calculating $lfp(f)$: simply do the finite number of calculations $f^{[0]}, \ldots, f^{[n_0]} = f^{n_0}(f^{[0]}) = lfp(f)$.

In general $\lambda$ cannot simply be taken to be a natural number and then we shall be content with a computable procedure for finding some $fix_\nu(f) \in L$ such that $fix_\nu(f) \sqsubseteq lfp(f)$; for pragmatic reasons $fix_\nu(f)$ should be “as close to” $lfp(f)$ as possible. An upper bound operator $\nabla : L \times L \rightarrow L$ is a not necessarily monotone function such that $l_1 \nabla l_2 \sqsupseteq l_1 \sqcup l_2$ for all $l_1, l_2 \in L$. Given a not necessarily increasing sequence $(l_n)_n$ we can then define

$$l_n^{\nabla} = \sqcap l_0$$
$$l_{n+1}^{\nabla} = l_n \nabla l_{n+1}$$

for all natural numbers $n$. It is easy to show that $(l_n^{\nabla})_n$ is in fact an increasing sequence. A widening operator $\nabla : L \times L \rightarrow L$ is an upper bound operator such that $(l_n^{\nabla})_n$ eventually stabilises (i.e. $\exists n_0 : \forall n \geq n_0 : l_n^{\nabla} = l_{n_0}^{\nabla}$) for all increasing sequences $(l_n)_n$. (A trivial and uninteresting example of a widening operator is given by $l_1 \nabla l_2 = \sqcup L$.) We can now define

$$f^{(0)} = \sqcap L$$

(the least element of $L$)

$$f^{(n+1)}_\nu = \begin{cases} 
  f^{(n)}_\nu & \text{if } f(f^{(n)}_\nu) \sqsubseteq f^{(n)}_\nu \\
  f^{(n)}_\nu \nabla f(f^{(n)}_\nu) & \text{otherwise}
\end{cases}$$

for all natural numbers $n$. One can show that $(f^{(n)}_\nu)_n$ eventually stabilises and that $fix_\nu(f) = \sqcup_{n} f^{(n)}_\nu$ equals $f^{(n_0)}_\nu$ for some $n_0$ and that $fix_\nu(f) \in Red(f)$. Hence $fix_\nu(f) \sqsubseteq lfp(f)$ and furthermore there is a computable procedure for calculating $fix_\nu(f)$: simply do the finite number of calculations $f^{(0)}_\nu, \ldots, f^{(n_0)}_\nu = fix_\nu(f)$. Examples of “good” widening operators can be found in the literature.

References


H.R. Nielson & F. Nielson