# A Survey of Static Program Analysis Techniques

Wolfgang Wögerer

Technische Universität Wien

October 18, 2005

**Abstract**

Computer program analysis is the process of automatically analysing the bahavior of computer programs. There are two main approaches in progam analysis: static program analysis and dynamic program analysis. In static analysis the programs are not executed but are analysed by tools to produce useful information. Static analysis techniques range from the most mundane (statistics on the density of comments, for instance) to the more complex, semantics-based analysis techniques. This text will give an overview of the more complex static program analysis techniques.

The main applications of program analysis are program optimization and program correctness. As far as correctness is concerned there is made a distinction between total correctness, where it is additionally required that the algorithm terminates, and partial correctness, which simply requires that if an answer is returned it will be correct. Since there is no general solution to the halting problem, a total correctness assertion is not solvable in general. In pratice the main goal is to use program analysis technology to build tools for validating industrial-sized software. However, two results of the theoretical computer science, namely the Rice's theorem [3] and the undecideabillity of the Halting Problem[1] [7] , show the limit of analysis methods. Generally precise program analysis methods are in exaustive need and processing time and/or memory space. Therefore the degree of precision must be determined application specific. Especially abstract interpretation offers good possibillities to handle this trade-off between precision and scalability.

In the following I will go deeper into the following topics:

- data flow analysis
- abstract interpretation
- symbolic analysis

[1]The undecidablity of the Halting Problem was proved by Alan Turing for the Turing Machine, which has unlimited memory. The proof uses this infinity to show that there can always be found one program where the algorithm in question cannot decide whether the program halts or not. Real computers do not have unlimited memory. Therefore, striktly speaking, the Halting Problem argument may not be used here. In this article, the undecideabillity of the Halting Problem shall represent the intractalbe problem of deciding whether a program on a real machine terminates.

# 1 Data Flow Analysis

Data flow analysis is a process for collecting run-time information about data in programs without actually executing them. Data flow anlaysis however does not make use of semenatics of operators. The semantics of the program language syntax is, however, implicity captured in the rules of the algorithms.

There are some common used terms when talking about data flow analysis that I want to define before going into details:

**basic block**: A basic block B is a sequence of consecutive instructions such that

1. control enters B only at its beginning;
2. control leaves B only at its end (under normal execution); and
3. control cannot halt or brach out of B except at its end

This implies that if any instruction in a basic block B is executed then all instruction in B are executed.

**control flow graph**: A control flow graph is an abstract representation of a (procedure or) program. Each node in the graph represents a basic block. Directed edges are used to represent jumps in the control flow. If a block has no incoming edge it is called an entry block, and if a block has no outgoing edge it is called an exit block.

**control flow path**: A control flow path is a path in the control flow graph that starts at an entry block and ends at an exit block. Normally, there extists more that one possible control flow paths, often the number of possible control flow paths is infinite because of the unpredictabillity of some loop bounds. One major task of the worst-case execution time analysis is to find the longest possible control flow path.

Data flow analysis is performed in two steps. In the first step so called gen/kill algorithms are used to collect the desired facts. The facts to be collected depend on the kind of data flow analysis in question. Gen/kill algorithms are very efficient because they just need one pass over the source code that is to be analysed and the only thing that those algorithms do is to check every statement if one or some of the variables in this statement should be added to the gen (generate) set or the kill set. The condition for adding variables to those set also depends on the kind of data flow analysis. The output of the gen/kill algorithm are $n$ gen set and $n$ kill sets where $n$ is the number of statements in the program. These sets are then used in the second set of the data flow analysis, the setting up and solving of equations.

When it comes to setting up the equations we can distinguish between forward analysis and backward analysis. In forward analysis the equations are set up in way that the information is transferred from the inital statements to the final statments, in backward analysis the information is transferred from the final

statments to the initial statements. To make this concrete in forward analysis the equations are set up according to the following scheme:

$$entry_\ell = \begin{cases} \emptyset & \text{if } \ell \text{ is an initial statement} \\ \bigcup exit_{\ell'} & \text{where } \ell' \text{ is a ancestor statement of } \ell \end{cases}$$

$$exit_\ell = entry_\ell \setminus kill_\ell \cup gen_\ell$$

And in backward analysis the equations are set up as follows:

$$entry_\ell = exit_\ell \setminus kill_\ell \cup gen_\ell$$

$$exit_\ell = \begin{cases} \emptyset & \text{if } \ell \text{ is an final statement} \\ \bigcup entry_{\ell'} & \text{where } \ell' \text{ is a ancestor statement of } \ell \end{cases}$$

Whether forward or backward analysis needs be used is given by the problem. Details can be found in [4].

We can apply data flow analysis for,amongst others, the following applications:

**Available Expression Analysis** (forward analysis): Here we are interested in expressions that are computed in more than one place in the program. If the expression, e.g. x op y, has been computed and there are no intervening kills of x and y, then the expression x op y may be substituted with the previous computation.

**Reaching Definitions Analysis** (forward analysis): The aim is here to determine for each program point which assignments may have been made and not overwritten when program execution reaches this point along some path. That way for example definitions can be found that are never used.

**Very Busy Expression Analysis** (backward analysis): We want to know for every exit of a basic block the expressions that, no matter what path is taken, are used. If the variables of the expression do not change until each usage then the expression is called busy and compiler can produce the code for evaluating that expression just once.

**Live Variable Analysis** (backward analysis): We want to know for variable $v$ of location $p$ in a program whether the value of $v$ at a previous location $p'$ could be used along some path in the flow graph starting at $p$. If so, we say $v$ is live at $p'$; otherwise v is dead at $p'$. This information is used for register allocation, e.g., after a value is computed in a register and it is dead at the end of the block, then it is not necessary to store that value.

Assume we have the following program and want to determine the live variables (the program language used is called WHILE and is defined in [5]):

```
[x:=2]1 ; [y:=4]2 ; [x:=1]3 ; (if [y>x]4 then [z:=y]5 else
[z:=y*y]6 ) ; [x:=z]7
```

The rule the produce the *gen* set: If statement $\ell$ is an assignment then $\text{gen}_\ell$ consists of all variables of the right hand side. If the statement is a condition that $gen_\ell$ consists of all varibles of that condition. Otherwise the set is empty.

The rule the produce the *kill* set: If statement $\ell$ is an assignment then $kill_\ell$ consists of the variable in the left hand side, otherwise the set is empty.

We therefore get the following sets:

| $\ell$ | $kill_\ell$ | $gen_\ell$ |
|---|---|---|
| 1 | {x} | $\emptyset$ |
| 2 | {y} | $\emptyset$ |
| 3 | {x} | $\emptyset$ |
| 4 | $\emptyset$ | {x,y} |
| 5 | {z} | {y} |
| 6 | {z} | {y} |
| 7 | {x} | {z} |

Applying the equation scheme for backward analysis we get the following equations:

$$
\begin{array}{ll}
entry_1 = exit_1 \setminus \{\text{x}\} & exit_1 = entry_2 \\
entry_2 = exit_2 \setminus \{\text{y}\} & exit_2 = entry_3 \\
entry_3 = exit_3 \setminus \{\text{x}\} & exit_3 = entry_4 \\
entry_4 = exit_4 \cup \{\text{x,y}\} & exit_4 = entry_5 \cup entry_6 \\
entry_5 = (exit_5 \setminus \{\text{z}\}) \cup \{\text{y}\} & exit_5 = entry_7 \\
entry_6 = (exit_6 \setminus \{\text{z}\}) \cup \{\text{y}\} & exit_6 = entry_7 \\
entry_7 = \{\text{z}\} & exit_7 = \emptyset
\end{array}
$$

Solving this equations is pretty easy since we just need to iteratively replace the sets that do not depend on other equations. So, we can replace the term $entry_7$ in the equation of $exit_6$ and $exit_5$. Then we can replace the term $exit_6$ in the equation of $entry_6$, then $exit_6$ in $entry_4$, and so on. This will result in the following solution of the live variable analysis:

$$
\begin{array}{ll}
entry_1 = \emptyset & exit_1 = \emptyset \\
entry_2 = \emptyset & exit_2 = \{\text{y}\} \\
entry_3 = \{\text{y}\} & exit_3 = \{\text{x,y}\} \\
entry_4 = \{\text{x,y}\} & exit_4 = \{\text{y}\} \\
entry_5 = \{\text{y}\} & exit_5 = \{\text{z}\} \\
entry_6 = \{\text{y}\} & exit_6 = \{\text{z}\} \\
entry_7 = \{\text{z}\} & exit_7 = \emptyset
\end{array}
$$

However, solving the equations is not generally as easy as that, as the following example illustrates:

The program

`(while [x>1]`$^1$ `do [skip]`$^2$ `) [x:=x+1]`$^3$

leads to the equations

$$\begin{array}{ll} entry_1 = exit_1 \cup \{\mathrm{x}\} & exit_1 = entry_2 \cup entry_3 \\ entry_2 = exit_2 & exit_2 = entry_1 \\ entry_3 = \{\mathrm{x}\} & exit_3 = \emptyset \end{array}$$

Now, this replacing procedure of the example above does not work anymore because there are too much dependencies within the equations. By continually replacing terms we would end up stuck with equations like:

$$\begin{array}{ll} entry_1 = entry_1 \cup \{\mathrm{x}\} & exit_1 = entry_2 \cup entry_3 \\ \boldsymbol{entry_2 = entry_2 \cup \{\mathrm{x}\}} & exit_2 = entry_1 \\ entry_3 = \{\mathrm{x}\} & exit_3 = \emptyset \end{array}$$

In fact by replacing in our first solution the set {x} by any superset of {x} is a solution to the equation system. What we are looking for, however, is the smallest soluation. One way to solve that problem is to find an algorithm that uses fixpoint construction, i.e. an algorithm that iteratively approximates the smallest solution until the smallest solution is found. This algorithm needs to fulfill two properties:

1. the solution found is really the smallest solution and
2. the algorithm always terminates.

The MFP (minimum fixed point) algorithm described in [4] fulfills both properties. It actually does not use the equations themselves to find a solution but the rules to construct those equations. The algorithm starts at the first statement of the program and produces the gen/kill sets. It then analysis the next statement where the produces gen/kill sets of the previous statements are transferred as an input. These gen/kill sets are now being altered according to the statement in question and are then forwarded to the next statement. In case of an if clause both branches are analysed seperately, where of course both get the same gen/kill sets as input. At the end of the branch the two sets are merged (analogously to construct an entry set with more that one immediate ancestor statements). In case of loops, this is where the fixed points come in, the loops are iteratively run through until the gen/kill sets do not change anymore.

Data flow analysis is a very efficent and feasable way of program analysis and is mainly used in compilers to create optimised code. For the purpose of detecting possible runtime errors or the calculation of the wort-case execution time of a program it is however not powerful enough. Many interesting pieces of information cannot be gatherd because data flow analysis does not make use of the semantic of the programming language's operators.

## 2 Abstract Interpretation

Abstract interpretation is a theory of semantics approximation. The idea of abstract interpretation is to create a new semantic of the programming language so that the semantic always terminates and the store for every program points contains a superset of the values that are possible in the actual semantic, for every possible input. Since in this new semantic a store does not contain a single value for a variable but a set (or interval) of possible values, the evaluation of boolean and arithmetic expressions must be redefined.

To make this method more feasable usually an abstract value domain is also defined. Therefore two functions need to be defined, one for mapping a concrete value to an (or a set of) abstract value(s) and one to map an abstract value to a (or set of) concrete values. It is obvious that by using an abstract domain information gets lost. But even without using an abstract domain information gets lost because the new semantic needs to always terminate. Loops in the new semantic must have a heuristic that determines when to set all variables in question to the value "all values possible". But there could have been a fixpoint in that loop would have been more precise. The framework of abstract interpretation was intrduced by Patrick Cousot and Radhia Cousot in [1]. Using this mathematical framework (together with its restrictions as to always use a lattice as abstract domain) it is relatively easy to logically show that the new semantic terminates and always gives a correct abstraction of the actual semantic.

Using that abstract semantics one can detect some possible semantic errors, like division by zero. We can also use that technique to verify that a program returns results within a certain range (if the program terminates! - this can of course not be proven since that problem is equivalent to the Halting Problem [7]).

In the following I will define an abstract semantic for the WHILE language (as defined in [5]). It is assumed that just variables with integer values are used and the abstract value domain is based on intervals.

The value domain is normally the most important desing issue when it comes th construction an abstract interpretation because it naturally greatly influences the other parts of the semantic. There are however some restrictions. The value domain must form a partially ordered set and need to support the greatest lower bound operator and the least upper bound operator for each tuple of the value domain. Also it must contain a least element and a top element. There must also exist a monotone function $\gamma : \tilde{D} \to D$, called concretisation function, that transforms an item of the abstract domain to the concrete domain, and there must be a second monotone function $\alpha : D \to \tilde{D}$ that performs the transformation in the opposite direction. In general $\gamma$ is not the inverse function of $\alpha$, since $\alpha$ uses some abstraction. Here is an example: Let $v \in D = \{1, 2, 7, 8\}$ then $\gamma(v) = [1..8]$ and $\alpha([1..8]) = \{1, 2, 3, 4, 5, 6, 7, 8\}$.

The formal definition of our concrete and abstract value domain is

$D = \mathcal{P}(\mathbb{N} \mid maxint < n < minint, n \in \mathbb{N}) \cup \bot \cup \top$
$\tilde{D} = \{[l..u] \mid l, u \in \mathbb{N} \text{ and } minint \leq l, u \leq maxint \text{ and } l \leq u\} \cup \bot_{int} \cup \top_{int}$

where $\bot$ and $\bot_{int}$ represent a not initialized variable value and $\top$ and $\top_{int}$ represent all possible values including the error value.

Both the abstract domain and the concrete domain need to be a partially ordered set with a least element and a top element. The ordering is defined as

$v_1 \sqsubseteq v_2 \iff v_1 \subset v_2$
$\bot \sqsubseteq v \quad \forall\, v \in D$
$v \sqsubseteq \top \quad \forall\, v \in D$

$[l_1, u_1] \sqsubseteq_{int} [l_2, u_2] \iff l_1 \leq l_2 \text{ and } u_1 \leq u_2$
$\bot_{int} \sqsubseteq \tilde{v} \quad \forall\, \tilde{v} \in \tilde{D}$
$\tilde{v} \sqsubseteq \top_{int} \quad \forall\, \tilde{v} \in \tilde{D}$

The greates lowest bound operators are definded as

$v \sqcap \top = \top \sqcap v = v$
$v_1 \sqcap v_2 = v_1 \cap v_2$
$v \sqcap \bot = \bot \sqcap v = \bot$

$\tilde{v} \sqcap_{int} \top_{int} = \top_{int} \sqcap_{int} \tilde{v} = \tilde{v}$
$[l_1..u_1] \sqcap_{int} [l_2..u_2] = \begin{cases} [max(l_1, l_2)..min(u_1, u_2)] & \text{if } max(l_1, l_2) \leq min(u_1, u_2) \\ \bot_{int} & \text{if } max(l_1, l_2) < min(u_1, u_2) \end{cases}$
$\tilde{v} \sqcap_{int} \bot_{int} = \bot_{int} \sqcap_{int} \tilde{v} = \bot_{int}$

The least upper bound operators are definded as

$v \sqcup \top = \top \sqcup v = \top$
$v_1 \sqcup v_2 = v_1 \cup v_2$
$v \sqcup \bot = \bot \sqcup v = \bot$

$\tilde{v} \sqcup_{int} \top_{int} = \top_{int} \sqcup_{int} \tilde{v} = \top_{int}$
$[l_1..u_1] \sqcup_{int} [l_2..u_2] = [min(l_1, l_2)..max(u_1, u_2)]$
$\tilde{v} \sqcup_{int} \bot_{int} = \bot_{int} \sqcup_{int} \tilde{v} = \tilde{v}$

The arthmetic operators of the WHILE language are now redefined as

$\top_{int}\ op\ \tilde{v} = \tilde{v}\ op\ \top_{int} = \top_{int}$ if $op \in \{\tilde{+}, \tilde{-}, \tilde{*}, \tilde{/}\}$, for any value $\tilde{v} \neq \bot_{int}$

$[l_1..u_1] \mathbin{\tilde{+}} [l_2..u_2] = \begin{cases} [l_1 + l_2..u_1 + u_2] & \text{if } minint \leq (l_1 + l_2),\ (u_1 + u_2) \leq maxint \\ \top_{int} & \text{otherwise} \end{cases}$
$[l_1..u_1] \mathbin{\tilde{-}} [l_2..u_2] = \begin{cases} [l_1 - l_2..u_1 - u_2] & \text{if } minint \leq (l_1 - u_2),\ (u_2 - l_2) \leq maxint \\ \top_{int} & \text{otherwise} \end{cases}$

$$[l_1..u_1] \tilde{*} [l_2..u_2] = \begin{cases} [min(V)..max(V)] & \text{if } minint \leq min(V),\ max(V) \leq maxint \\ \quad \text{where V} = \{l_1 * l_2, l_1 * u_2, u_1 * l_2, u_1 * u_2\} & \\ \top_{int} & \text{otherwise} \end{cases}$$

$$[l_1..u_1] \tilde{/} [l_2..u_2] = \begin{cases} [min(V)..max(V)] & \text{if } minint \leq min(V),\ max(V) \leq maxint \\ \quad \text{and } (u_2 < 0 \text{ or } l_2 > 0) \text{ where V} = \{l_1/l_2, l_1/u_2, u_1/l_2, u_1/u_2\} & \\ \top_{int} & \text{otherwise} \end{cases}$$

$\bot_{int}\ op\ \tilde{v} = \tilde{v}\ op\ \bot_{int} = \bot_{int}$ if $op \in \{\tilde{+}, \tilde{-}, \tilde{*}, \tilde{/}\}$, for any value $\tilde{v}$

As a next step the definition of evaluation of comparisions for $<$, $>$, $>=$, $<=$, $=$ are defined. When comparing sets it is possible that some combinations of elements evaluate to true and others to false. In that case both branches that follow the condition must be taken. At the location where both braches meet, all variables have to be merged by calculation the least upper bound for each variable pair.

When it comes to loops, the loop is iterated until the conditions evaluates to false or the heuristic of the abstract semantic decides that no fixed point will be found in acceptable time and sets all variables that can be changed in the loop to the top element.

Abstract interpretation is a very powerful program analysis method. It uses information on the programming language's semantic and can detect possible runtime errors, like division by zero or variable overflow. Since abstract interpretation can be computationally very expensive it must be taken care choose an appropriate value domain and appropiate heuristic for loop termination to ensure feasabillity.

# 3 Symbolic Analysis

Symbolic analysis is a static analysis method for reasoning about program values that may not be constant. It aims to derive a precise mathematical characterisation of the computations and can be seen as a compiler that translates a program into a different language whereas this language consists of symbolic expressions and symbolic recurrences.

Computer algebra systems (such as Axiom, Derive, Macsyma, Maple, Mathematica, MuPAD, and Reduce) play an important role in supporting this technique since the quality of the final results depend significantly on smart algebraic simplification methods.

The properties and the idea of symbolic analysis might be best illustrated by an example. The following example shows a sequence of statements written in some standard programming language. Below each statement there is the program context for this statement. The result of an analysis is the program context of all program's exit points. The program context constist of tree parts: The state ($s$), the state condition ($t$) and the path condition ($p$). The state is a set of varialbe/value pairs whereas there is for each variable exacly one value. Uninitialised values are denoted by $\perp$ and the reading from a stream is symbolized by $\nabla_i$. In programs without branches (like the example below) just the state is of interest. The path condition and the state condtion will be explained later.

---

$\ell_0$: `int a, b;`
$[s_0 = \{a = \perp, b = \perp\}, t_0 = true, p_0 = true]$
$\ell_1 : read(a);$
$[s_1 = \{a = \nabla_1, b = \perp\}, t_1 = true, p_1 = true]$
$\ell_2 : read(b);$
$[s_2 = \{a = \nabla_1, b = \nabla_2\}, t_2 = true, p_2 = true]$
$\ell_3 : a = a + b;$
$[s_3 = \{a = \nabla_1 + \nabla_2, b = \nabla_2\}, t_3 = true, p_3 = true]$
$\ell_4 : b = a - b;$
$[s_4 = \{a = \nabla_1 + \nabla_2, b = (\nabla_1 + \nabla_2) - \nabla_2\}, t_4 = true, p_4 = true]$
$\ell_5 : a = a - b;$
$[s_5 = \{a = (\nabla_1 + \nabla_2) - ((\nabla_1 + \nabla_2) - \nabla_2), b = (\nabla_1 + \nabla_2) - \nabla_2\},$
$t_5 = true, p_5 = true]$

---

*Example 1.1: simple symbolic analysis*

As mentioned above the result of the analysis is given by the program contexts of the program's exit points. In the example above this comes down to the state $s_5$. However without simplifying the algebraic expressions of the two state variables $a$ and $b$ there will be no useful information. Therefore, let's now simplify:

$$
\begin{aligned}
a &= (\nabla_1 + \nabla_2) - ((\nabla_1 + \nabla_2) - \nabla_2) \\
&= \nabla_1 + \nabla_2 - \nabla_1 - \nabla_2 + \nabla_2 \\
&= \nabla_2
\end{aligned}
$$

$$
\begin{aligned}
b &= (\nabla_1 + \nabla_2) - \nabla_2 \\
&= \nabla_1 + \nabla_2 - \nabla_2 \\
&= \nabla_1
\end{aligned}
$$

Therefore, the analysis shows that the program above does nothing else then to assign the first input to varialbe $b$ and the second input to variable $a$.

Let's now take a look at a program that contains conditional statements:

$\ell_0$: `int a, b;`
$\quad [s_0 = \{a = \bot, b = \bot\}, t_0 = true, p_0 = true]$
$\ell_1 : read(a);$
$\quad [s_1 = \{a = \nabla_1, b = \bot\}, t_1 = t_0, p_1 = p_0]$
$\ell_2 : if(a < 0)\{$
$\quad [s_2 = s_1, t_2 = t_1, p_2 = (p_1 \wedge \nabla_1 < 0)]$
$\ell_3 : \quad b = -2 * a;$
$\quad [s_3 = \delta(s_2; b = -2 * \nabla_1), t_3 = t_2, p_3 = p_2]$
$\ell_4 : \quad a = -b;$
$\quad [s_4 = \delta(s_3; a = 2 * \nabla_1), t_4 = t_3, p_4 = p_3]$
$\ell_5 : \} else \{$
$\quad [s_5 = s_1, t_5 = t_1, p_5 = (p_1 \wedge \nabla_1 \geqq 0)]$
$\ell_6 : \quad b = 2 * a;$
$\quad [s_6 = \delta(s_5; b = 2 * \nabla_1), t_6 = t_5, p_6 = p_5]$
$\ell_7 : \}$
$\quad [s_7 = \delta(s_1; a = \tilde{a}, b = \tilde{b}),$
$\quad t_7 = \gamma(\nabla_1 < 0; \tilde{a} = 2 * \nabla_1, \tilde{b} = -2 * \nabla_1; \tilde{a} = \nabla_1, \tilde{b} = 2 * \nabla_1),$
$\quad p_7 = p_4 \wedge p_6]$
$\ell_8 : if(a >= 0)\{$
$\quad [s_8 = s_7, t_8 = t_7, p_8 = p_7 \wedge \nabla_1 > 0] =$
$\quad [s'_8 = \{a = \nabla_1, b = 2 * \nabla_1\}, t'_8 = true, p'_8 = \nabla_1 < 0]$
$\ell_9 : \quad a = 2 * a;$
$\quad [s_9 = \delta(s'_8; a = 2 * \nabla_1), t_9 = t'_8, p_9 = p'_8]$
$\ell_{10} : \}$
$\quad [s_{10} = \delta(s_7; a = 2 * \nabla_1),$
$\quad t_{10} = \gamma(\nabla_1 < 0; \tilde{b} = -2 * \nabla_1; \tilde{b} = 2 * \nabla_1),$
$\quad p_{10} = true]$

*introduced functions:*

$\delta$ ... this function is used to avoid writing the whole set of variables each time. It states that that resulting set is equal to the first parameter except the variable/value pairs in the second parameter.

$\gamma$ ... this function consists of three parameters. This first paramter is a condition. This condition determines whether the list of variable/value-pairs in the second or the third parameter shall be true. More formally:
$\gamma(cnd; x_1 = e_1, ..., x_k = e_k; x_1 = f_1, ..., x_k = f_k) =$
$(cnd \wedge x_1 = e1 \wedge ... \wedge x_k = e_k) \vee (\neg cnd \wedge x_1 = f_1 \wedge ... \wedge x_k = f_k)$

*Example 1.2: symbolic analysis with conditional statements*

This example illustrates the use of the state condition ($t$) and the path condition ($p$). The state condition is a logic formula for describing assumptions about the variable values. These assuptions can either result from the preceding pramming instructions (p. e. in $\ell_7$ the state of variable $a$ equals $2 * \nabla_1$ assuming we took the if-branch in $\ell_2$) or the assumptions can be brought in from "outside" (i. e. by the person analysing the programing), p. e. in the upcoming example about loops we assume that variable $b$ is greater than zero when entering $\ell_1$.

The path condition is a logic formula that codes the condition that this program point is reached. If we find a contradiction in a path condition, we have found dead code. One now might think that a program point $k$ with $p_k = true$ will be reached in any case. While this is true for our model, it is not always true in reality. An overflow of an integer variable for example could make our prgram crash. When "compiling" our programming language source code to our model of symbolic analysis we loose some implementation details.

We could use the result from the example above to optimize the program. Since the only exit point in the program context of $\ell_{10}$ we can now retransform this program context to our source programming language:

$\ell_1 : read(a);$
$\ell_2 : a = 2 * a;$
$\ell_3 : if(a < 0)\{$
$\ell_4 : \quad b = -a;$
$\ell_5 : \} \; else \; \{$
$\ell_6 : \quad b = a;$
$\ell_7 : \}$

The quality of symbolic analysis depends critically on the ability to analyse loops. The following example should on the one hand illustrate the power of symbolic analysis what precision is concerned. On the other hand the example should give an idea of the difficulties in achiving this precision:

---

$[s_0 = \{a = \tilde{a}, b = \tilde{b}, c = \nabla_1, d = \tilde{d}\},$
$\; t_0 = (\tilde{a} > 0 \wedge \tilde{b} > 0 \wedge \tilde{d} > 0),$
$\; p_0 = true]$
$\ell_1 : while(a < b)\{$
$\quad [s_1 = \{a = \underline{a}, b = \tilde{b}, c = \underline{c}, d = \tilde{d}\}, t_1 = t_0, p_1 = \underline{a} \leqq b]$
$\ell_2 : \quad c = 2 * c;$
$\quad [s_2 = \delta(s_1; c = 2 * \underline{c}), t_2 = t_1, p_2 = p_1]$
$\ell_3 : \quad a = a + d;$
$\quad [s_3 = \delta(s_2; a = \underline{a} + d;), t_3 = t_2, p_3 = p_2]$
$\ell_4 : \}$
$[s_4 = \delta(s_0; a = \mu(a, s_0, [s_3, t_3, p_3]), c = \mu(c, s_0, [s_3, t_3, p_3])),$
$\; t_4 = t_0, p_4 = p_0] =$
$[s'_4 = \delta(s_0; a = d * \lceil \frac{b+1}{d} \rceil, c = \nabla_1 * 2^{\lceil \frac{b+1}{d} \rceil - 1}),$
$\; t'_4 = t_4, p'_4 = p_4]$

*introduced functions:*
$\mu(v, s, c)$ ... is the recurrence function. $v$ is the variable we want to find a closed form for. $s$ is the state that contains the initial values. $c$ is the program context of the last statement within the loop.

---

*Example 1.3: symbolic analysis and loops*

The underlined variables are loop variants, i. e. they may be different per iteration. One way to find a closed form for this loop is to reformulate the loop as an mathematical recursion. We therefore get rid of all loop variants. Instead we get indexed variables and their corresponding recursion formulas. By intelligently transforming these recursion formulas we can find the closed form.

Both tasks, i. e. the transformation of the loop to an recursion and the seach for an closed form for the recursion, are tough tasks whereof detailed explaination is beyond scope of this article. To give an impression how the loop problem can be approached concretly I will give a solution for the above example. Let us assume that we have found an algorithm that can transform the loop to the recursion formulae (1) to (4).

$$a(0) = b \tag{1}$$

$$a(k+1) = a(k) + d, k \geqq 0 \tag{2}$$

$$c(0) = \nabla_1 \tag{3}$$

$$c(k+1) = 2 * c(k), k \geqq 0 \tag{4}$$

Because of the simplicity of the example we can intiutively find the transformations for the closed forms. Let's first consider variable $a$. In each recursion step the only change to the previous value is the addition of $b$ which is also the starting value. We can therefore transform equation (1) und (2) to

$$a(k) = b * (k+1) \tag{5}$$

Similary we can find the closed form for variable $c$. Its value is doubled each step. And its starting value is $\nabla_1$. We can therefore transform (3) and (4) to

$$c(k) = \nabla_1 * 2k \tag{6}$$

The next step is now to find the last iteration. With $a(n) > b$ (recurrence condition) the last iteration z can be determined

$$z = min\{k|a(k) > b\} \tag{7}$$

$$= min\{k|d * (k+1) > b\} \tag{8}$$

$$= \left\lceil \frac{b+1}{d} \right\rceil - 1 \tag{9}$$

by subistituting z for k in (5) and (6) we get

$$c = \nabla_1 * 2^{\lceil \frac{b+1}{d} \rceil - 1} \tag{10}$$

$$a = d * \left\lceil \frac{b+1}{d} \right\rceil \tag{11}$$

When used in optimizing compilers symbolic anlysis can lead to very good results, both in the code size and in run-time behaviour. Symbolic analysis is also applicable in worst-execution time analysis, since unpredictable loops can be transformed in a predictable sequence. However, there are many loops that cannot be transformed in a closed form.

# 4 Conclusion

Data flow analysis is a special type of program analysis that is typically based on an abstract semantics of the program. Data flow analysis algorithms are quite easy to construct and usually computationally feasable.

Abstract interpretation is a generic framework for the construction of program analysis techniques. Every kind of program anlaysis can be seen as an instance of abstract interpretation.

Symbolic analyis is a technique to derive precise characterisations of program properties in a parameteric way.

# References


[1] Patrick Cousot, Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 238–252, Los Angeles, California, 1977. ACM Press, New York.

[2] Jan Gustafsson. Analysing Execution-Time of Object-Oriented Programs Using Abstract Interpretation. Uppsala University, 2000

[3] Moret, Bernard M. E.: The theory of computation. Addison-Wesley, 1998. ISBN 0-201-25828-5

[4] Nielson, Flemming : Principles of program analysis. - Berlin [u.a.] : Springer, 1999

[5] Hanne Riis Nielson and Flemming Nielson: Semantics with Applications: A Formal Introduction, revised edition, 1999.

[6] Scholz, Bernhard : Symbolic analysis of programs and its applications, 2001

[7] Turing, A.M. 1936 : On Computable Numbers, with an Application to the Entscheidungsproblem. Proceedings of the London Mathematical Society, Series 2, 42 (1936-37), pp.230-265.