Software Verification

Sebastian Nanz

Lecture 11: Abstract Interpretation
Plan for today's lecture

- In the first part we discuss program slicing as another example of an application of data flow analysis.

- In the second part we discuss abstract interpretation, a general framework for expressing program analyses.
Program Slicing
Program slicing

1. sum := 0
2. prod := 1
3. i := 0
4. while i < y do
5.   sum := sum + x
6.   prod := prod * x
7.   i := i + 1
8. end
9. print(sum)
10. print(prod)

"What program statements potentially affect the value of variable 'sum' at line 8 of the program?"
Program slicing

- Program slicing provides an answer to the question

  "What program statements potentially affect the value of variable x at program point l?"

- The resulting program statements are called the program slice.
- The pair \((x, l)\) is called the **slicing criterion**.
- An observer focusing on the slicing criterion (i.e. only observing the value of x at program point l) cannot distinguish a run of the program from the run of its slice.
Applications of program slicing

- **Debugging**: Slicing lets the programmer focus on the program part relevant to a certain failure, which might lead to quicker detection of a fault.

- **Testing**: Slicing can minimize test cases, i.e. find the smallest set of statements that produce a certain failure (good for regression testing).

- **Parallelization**: Slicing can determine parts of the program which can be computed independently of each other and can thus be parallelized.
Classification

- **Static slicing vs. dynamic slicing**
  - Static: general, not considering a particular input
  - Dynamic: computed for a fixed input, therefore smaller slices can be obtained

- **Backward slicing vs. forward slicing**
  - Backward: "Which statements affect the value a certain variable?"
  - Forward: "Which statements are affected by the value of a certain variable?"

- In the following we present an algorithm for static backward slicing.
For variable $x$ and program point $l$, a **backward slice** $S$ of program $P$ with respect to slicing criterion $(x, l)$ is any executable program with the following properties:

1. $S$ can be obtained by deleting zero or more statements from $P$.
2. If $P$ halts on input $I$, then the value of $x$ at program point $l$ is the same in $P$ and in $S$ every time program point $l$ is executed.
Slicing algorithm

- We present a slicing algorithm for static backward slicing.
- Many different approaches, we show one that constructs a program dependence graph (PDG).
- A PDG is a directed graph with two types of edges:
  - Data dependencies: given by data-flow analysis
  - Control dependencies: program point \( l \) is control-dependent on program point \( l' \) if
    1. \( l' \) labels the guard of a control structure
    2. the execution of \( l \) depends on the outcome of the evaluation of the guard at \( l' \)
Control flow graph of the example program

1. \[\text{sum} := 0\]
2. \[\text{prod} := 1\]
3. \[i := 0\]
4. \[i < y\]
5. \[\text{sum} := \text{sum} + x\]
6. \[\text{prod} := \text{prod} \times x\]
7. \[i := i + 1\]
8. \[\text{print(sum)}\]
9. \[\text{print(prod)}\]
Example: Program dependence graph

1. Data dependence subgraph

\[
\begin{align*}
\text{[sum := 0]}_1 & \quad \text{[prod := 1]}_2 \\
\text{[i := 0]}_3 & \quad \rightarrow \quad [i < y]_4 \\
\text{[sum := sum + x]}_5 & \quad \text{[prod := prod * x]}_6 \\
\text{[i := i + 1]}_7 & \\
\text{[print(sum)]}_8 & \quad \text{[print(prod)]}_9 \\
\end{align*}
\]

----> \{ (l, l') \mid l \in \bigcup_{x \text{ used in block } l'} \text{UD}(x, l') \text{ where } l' \text{ labels a block} \}

(self-loops are omitted)
Example: Program dependence graph

2. **Control dependence subgraph**

(1) Edge from special node ENTRY to any node not within any control structure (such as while, if-then-else)

(2) Edge from any guard of a control structure to any statement within the control structure
Example: Computing the program slice

Slicing using the PDG:

1. Take as initial node the one given by the slicing criterion
2. Include all nodes which the initial node transitively depends upon (use both data- and control-dependencies)
Abstract Interpretation

Introduction
One framework to rule them all

- In the past lecture we have introduced a particular style of program analysis: data flow analysis.

- For these types of analyses, and others, a main concern is correctness: how do we know that a particular analysis produces sound results (does not forget possible errors)?

- In the following we discuss abstract interpretation, a general framework for describing program analyses and reasoning about their correctness.
Main ideas: Concrete computations

- An ordinary program describes computations in some **concrete domain** of values.
  - **Example:** program states that record the integer value of every program variable.
    \[
    \sigma \in \text{State} = \text{Var} \rightarrow \mathbb{Z}
    \]

- Possible computations can be described by the **concrete semantics** of the programming language used.
Main ideas: Abstract computations

- Abstract interpretation of a program describes computation in a different, abstract domain.
- **Example:** program states that only record a specific property of integers, instead of their value: their sign, whether they are even/odd, or contained in \([-32768, 32767]\) etc.

\[ \sigma \in \text{AbstractState} = \text{Var} \rightarrow \{\text{even, odd}\} \]

- In order to obtain abstract computations, an abstract semantics for the programming language has to be defined.
- Abstract interpretation provides a framework for proving that the abstract semantics is sound with respect to the concrete semantics.
Introductory example: Expressions

A little language of expressions

Syntax
\[ e ::= n \mid e \ast e \]

Concrete semantics
\[ C[n] = n \]
\[ C[e \ast e] = C[e] \cdot C[e] \]

Example
\[ C[-3 \ast 2 \ast -5] = C[-3 \ast 2] \cdot C[-5] = C[-3 \ast 2] \cdot (-5) = \ldots = 30 \]
Introductory example: Abstraction

Assume that we are not interested in the value of an expression but only in its sign:

- Negative: -
- Zero: 0
- Positive: +

Abstract semantics

\[ A[n] = \text{sign}(n) \]

\[ A[e \times e] = A[e] \otimes A[e] \]

Example

\[ A[-3 \times 2 \times -5] = A[-3 \times 2] \otimes A[-5] = A[-3 \times 2] \otimes (-) = \ldots = (-) \otimes (+) \otimes (-) = (+) \]
Introductory example: Soundness

- We want to express that the abstract semantics correctly describes the sign of a corresponding concrete computation.
- For this we first link each concrete value to an abstract value:

Representation function

\[ \beta : \mathbb{Z} \rightarrow \{-, 0, +\} \]

\[ \beta(n) = \begin{cases} 
- & \text{if } n < 0 \\
0 & \text{if } n = 0 \\
+ & \text{if } n > 0 
\end{cases} \]
Introductory example: Soundness

Conversely, we can also link abstract values to the set of concrete values they describe:

**Concretization function**

\[ \gamma : \{-, 0, +\} \to \wp(\mathbb{Z}) \]

\[ \gamma(s) = \begin{cases} 
{n | n < 0} & \text{if } s = - \\
\{0\} & \text{if } s = 0 \\
\{n | n > 0\} & \text{if } s = +
\end{cases} \]

**Soundness** then describes intuitively that the concrete value of an expression is described by its abstract value:

\[ \forall e. C[e] \subseteq \gamma(A[e]) \]
Extending the language

Syntax
\[ e ::= n \mid e \ast e \mid e + e \mid -e \]

Abstract semantics
\[ A[n] = \text{sign}(n) \]
\[ A[-e] = \ominus A[e] \]

Observation: The abstract domain \{-,0,+\} is not closed under the interpretation of addition.
Extending the abstract domain

We have to introduce an additional abstract value:

\[ \top \quad "top" \quad - \quad \text{(any value)} \]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>T</td>
</tr>
<tr>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td>T</td>
</tr>
</tbody>
</table>
The new abstract domain

We can extend the concretization function to the new abstract domain \{-,0,+, \top, \bot\} (add \bot for completeness):

\[
\gamma(\top) = \mathbb{Z} \quad \gamma(\bot) = \emptyset
\]

We obtain the following structure when drawing the partial order induced by

\[
a \leq b \text{ iff } \gamma(a) \subseteq \gamma(b)
\]

How is such a structure called?

A complete lattice (see lecture on program analysis).
Partially ordered sets (recap)

For any analysis, we are interested in expressing that one analysis result is "better" (more precise) than another. In other words, we want the analysis domain to be partially ordered.

A partial ordering is a relation $\sqsubseteq$ that is

- reflexive: $\forall d : d \sqsubseteq d$
- transitive: $\forall d_1, d_2, d_3 : d_1 \sqsubseteq d_2$ and $d_2 \sqsubseteq d_3$ imply $d_1 \sqsubseteq d_3$
- anti-symmetric: $\forall d_1, d_2 : d_1 \sqsubseteq d_2$ and $d_2 \sqsubseteq d_1$ imply $d_1 = d_2$

A partially ordered set $(D, \sqsubseteq)$ is a set $D$ with a partial ordering $\sqsubseteq$.

Examples: Real numbers $(\mathbb{R}, \leq)$, power sets $(\mathcal{P}(S), \subseteq)$, ...
Complete lattices

We are aiming for a specific kind of partially ordered set with even nicer properties: **complete lattices**.

- \( d \in D \) is an **upper bound** of \( Y \) if \( \forall d' \in Y : d' \sqsubseteq d \)
- A **least upper bound** \( d \) of \( Y \) is an upper bound of \( Y \) that satisfies \( d \sqsubseteq d_0 \) whenever \( d_0 \) is another upper bound of \( Y \)
- A **complete lattice** is a partially ordered set \( (D, \sqsubseteq) \) such that each subset \( Y \) has a least upper bound \( \bigcup Y \) (and a greatest lower bound \( \bigcap Y \))

**Example:** Power sets
Construction of complete lattices

- If we know some complete lattices, we can construct new ones by combining them.
- Such constructions become important when designing new analyses with complex analysis domains.

**Example:** Total function space

Let \((D_1, \sqsubseteq_1)\) be a partially ordered set and let \(S\) be a set. Then \((D, \sqsubseteq)\), defined as follows, is a complete lattice:

- \(D = S \rightarrow D_1\) ("space of total functions")
- \(f \sqsubseteq f'\) iff \(\forall s \in S : f(s) \sqsubseteq_1 f'(s)\) ("point-wise ordering")
Abstract Interpretation

Foundations
The framework of abstract interpretation

- Starting from a concrete domain $C$, define an abstract domain $(A, \sqsubseteq)$, which must be a complete lattice.
- Define a representation function $\beta$ that maps a concrete value to its best abstract value
  
  $$\beta : C \rightarrow A$$

- From this we can derive the concretization function $\gamma$
  
  $$\gamma : A \rightarrow \mathcal{P}(C)$$
  
  $$\gamma(a) = \{ c \in C \mid \beta(c) \sqsubseteq a \}$$

- and abstraction function $\alpha$ for sets of concrete values
  
  $$\alpha : \mathcal{P}(C) \rightarrow A$$
  
  $$\alpha(C) = \bigsqcup \{ \beta(c) \mid c \in C \}$$
Galois connections

- The following properties of $\alpha$ and $\gamma$ hold:

**Monotonicity**

(1) $\alpha$ and $\gamma$ are monotone functions

**Galois insertion**

(2) $a = \alpha(\gamma(a))$ for all $a \in A$

(3) $C' \subseteq \gamma(\alpha(C'))$ for all $C' \subseteq C$

- Galois insertion (or, less precise: Galois connection): This property means intuitively that the functions $\alpha$ and $\gamma$ are "almost inverses" of each other.
Induced Operations

- A Galois connection can be used to induce the abstract operations from the concrete ones.

```
\[ \mathcal{P}(C) \xrightarrow{\text{op}} \mathcal{P}(C) \]
```

\[ A \xrightarrow{\alpha \circ \text{op} \circ \gamma} A \]

abstract execution

```
\[ \mathcal{P}(C) \xrightarrow{\text{op}} \mathcal{P}(C) \]
```

concrete execution

- We can show that the induced operation \( \text{op} = \alpha \circ \text{op} \circ \gamma \) is the most precise abstract operation in this setting.

- The induced operation might not be computable. In this case we can define an upper approximation \( \text{op}^\# \), \( \text{op} \subseteq \text{op}^\# \), and use this as abstract operation.
Abstract Interpretation

Worked Example
Abstract interpretation for programs

- In the introductory example, we have abstracted the value of expressions only.
- We now move on to the abstraction of program states.
- For this we first introduce the notion of a collecting semantics, which associates with each program point the set of all states that can arise at this point during execution.
The collecting semantics

We assume the state of a program to be modeled as:

\[ \sigma \in \text{State} = \text{Var} \rightarrow \mathbb{Z} \]

We will use the following notation for function update:

\[ \sigma[x \mapsto k](y) = \begin{cases} k & \text{if } x = y \\ \sigma(x) & \text{otherwise} \end{cases} \]

We construct the collecting semantics as a function which gives for every program label the set of all possible states.

\[ C : \text{Labels} \rightarrow \mathcal{P}(\text{State}) \]
Rules of the collecting semantics

\[ C_{l'} = \{ \sigma[x \mapsto n] \mid \sigma \in C_l \text{ and } C[e] \sigma = n \} \]

\[ C_{l_{\text{true}}} = \{ \sigma \mid \sigma \in C_l \text{ and } C[b] \sigma = \text{true} \} \]

\[ C_{l_{\text{false}}} = \{ \sigma \mid \sigma \in C_l \text{ and } C[b] \sigma = \text{false} \} \]

\[ C_l = C_{l_1} \cup C_{l_2} \]

Note: In difference to the lecture on program analysis, labels are not on blocks, but on edges.
Example: Collecting semantics

Assume $x > 0$.

$$C_1 = \{ \sigma \mid \sigma(x) > 0 \}$$

$$C_2 = \{ \sigma[y \mapsto 1] \mid \sigma \in C_1 \} \cup \{ \sigma[x \mapsto \sigma(x) - 1] \mid \sigma \in C_4 \}$$

$$C_3 = C_2 \cap \{ \sigma \mid \sigma(x) \neq 0 \}$$

$$C_4 = \{ \sigma[y \mapsto \sigma(x) \cdot \sigma(y)] \mid \sigma \in C_3 \}$$

$$C_5 = C_2 \cap \{ \sigma \mid \sigma(x) = 0 \}$$
Solving the equations

- The equation system we obtain has variables $C_1, \ldots, C_5$ which are interpreted over the complete lattice $\mathcal{P}(\text{State})$.
- We can express the equation system as a monotone function $F : \mathcal{P}(\text{State})^5 \to \mathcal{P}(\text{State})^5$
  \[
  F(C_1, \ldots, C_5) = (\{\sigma \mid \sigma(x) > 0\}, \ldots, C_2 \cap \{\sigma \mid \sigma(x) = 0\})
  \]
- Using Tarski's Fixed Point Theorem, we know that a least fixed point exists (see lecture on program analysis).
- Trick from lecture on program analysis: The least fixed point can be computed by repeatedly applying $F$, starting with the bottom element $\bot = (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$ of the complete lattice until stabilization.
  \[
  F(\bot) \sqsubseteq F(F(\bot)) \sqsubseteq \ldots \sqsubseteq F^n(\bot) = F^{n+1}(\bot)
  \]
Tarski's Fixed Point Theorem (recap)

Monotone function
A function $F : D \rightarrow D$ is called monotone over $(D, \sqsubseteq)$ if
\[ d \sqsubseteq d' \text{ implies } F(d) \sqsubseteq F(d') \quad \text{for all } d, d' \in D \]

Fixed point
Assume $F : D \rightarrow D$. A value $d \in D$ such that $F(d) = d$ is called a fixed point of $F$.

Tarski's Fixed Point Theorem
Let $(D, \sqsubseteq)$ be a complete lattice and let $F : D \rightarrow D$ be a monotone function. Then the set of all fixed points of $F$ is a complete lattice with respect to $\sqsubseteq$.

In particular, $F$ has a least and a greatest fixed point.
Example: Fixed Point Computation

\[ y := 1 \]
\[ x \neq 0 \]
\[ y := x \cdot y \]
\[ x := x - 1 \]

\[ C_1 = \{ \sigma \mid \sigma(x) > 0 \} \]
\[ C_2 = \{ \sigma[y \mapsto 1] \mid \sigma \in C_1 \} \cup \{ \sigma[x \mapsto \sigma(x) - 1] \mid \sigma \in C_4 \} \]
\[ C_3 = C_2 \cap \{ \sigma \mid \sigma(x) \neq 0 \} \]
\[ C_4 = \{ \sigma[y \mapsto \sigma(x) \cdot \sigma(y)] \mid \sigma \in C_3 \} \]
\[ C_5 = C_2 \cap \{ \sigma \mid \sigma(x) = 0 \} \]
The ascending chain condition

- Using a concrete semantics the solution is not computable in general.

- The reason for that is the fact that the value space is unbounded, it contains infinite ascending chains:

  \((l_n)_n\) is such that \(l_1 \subseteq l_2 \subseteq l_3 \subseteq \cdots\),
  
  but there exists no \(n\) such that \(l_n = l_{n+1} = \cdots\)

- If we replace it with an abstract space that is finite (or does not possess infinite ascending chains), then the computation is guaranteed to terminate.

- In general, we want an abstract domain to satisfy the ascending chain condition, i.e. each ascending chain eventually stabilises:

  if \((l_n)_n\) is such that \(l_1 \subseteq l_2 \subseteq l_3 \subseteq \cdots\),
  
  then there exists \(n\) such that \(l_n = l_{n+1} = \cdots\)
Example: Abstract interpretation

Assume $x > 0$. Use the abstract domain for sign analysis.

\[
\begin{align*}
A_1 &= [x \mapsto +, y \mapsto T] \\
A_2 &= A_1[y \mapsto +] \sqcup \\
A_4[x \mapsto A_4(x) \ominus +] \\
A_3 &= A_2 \\
A_4 &= A_3[y \mapsto A_3(x) \otimes A_3(y)] \\
A_5 &= A_2 \sqcap [x \mapsto 0, y \mapsto T]
\end{align*}
\]
Abstract Interpretation

Widening
Range analysis

➢ To introduce the notion of widening, we have a look at range analysis, which provides for every variable an over-approximation of its integer value range.

➢ We are left with the task of choosing a suitable abstract domain: the interval lattice suggests itself.

\[ \text{Interval} = \{ \bot \} \cup \{ [x,y] \mid x \leq y, x \in \mathbb{Z} \cup \{\infty\}, y \in \mathbb{Z} \cup \{\infty\} \} \]
Example

Consider the following program:

\[
\begin{array}{c}
1 \quad [x \mapsto \top] \\
\downarrow \\
[x:=1] \\
\downarrow \\
2 \quad [x \mapsto [1,1]] \sqcup [x \mapsto [2,2]] = [x \mapsto [1,2]] \\
\downarrow \\
[x \leq n] \\
\downarrow \\
3 \quad [x \mapsto [1,1]] \\
\downarrow \\
[x:=x+1]
\end{array}
\]

At program point 2, the following sequence of abstract states arises: \([x \mapsto [1,1]], [x \mapsto [1,2]], [x \mapsto [1,3]], \ldots\)

Consequence: The analysis never terminates (or, if \(n\) is statically known, converges only very slowly).
The reason for the non-termination is that the interval lattice contains infinite ascending chains.

Trick: We redefine the join operator of the lattice to jump to the extremal value more quickly.

Before: \([1,1] \sqcup [2,2] = [1,2]\)  
Now: \([1,1] \triangledown [2,2] = [1,\infty]\)
**Widening**

A *widening* \(\nabla: D \times D \to D\) on a partially ordered set \((D, \sqsubseteq)\) satisfies the following properties:

1. For all \(x, y \in D\). \(x \sqsubseteq x \nabla y\) and \(y \sqsubseteq x \nabla y\)
2. For all ascending chains \(x_1 \sqsubseteq x_2 \sqsubseteq x_3 \sqsubseteq \cdots\) the ascending chain \(y_1 = x_1 \sqsubseteq y_2 = y_1 \nabla x_2 \sqsubseteq \cdots \sqsubseteq y_{n+1} = y_n \nabla x_{n+1}\) eventually stabilizes.

- Widening is used to accelerate the convergence towards an upper approximation of the least fixed point.
Example (continued)

- Assume we have a widening operator $\nabla$ that is defined such that $[1,1] \nabla [2,2] = [1, +\infty]$

- The analysis converges quickly.
Reading


Chapter 1: Section 1.5
Chapter 4 (advanced material)