Examples of Point-wise Abstraction

A point-wise abstraction is obtained by identifying a function $A : C$ mapping individual concrete states to individual abstract states. We construct ordered abstract domains by forming sets of individual states.

We consider abstraction mappings which are presented by a set of equations

\[ \{u_n, \ldots, u_1\} = \Lambda \]

or more compactly,

\[ ((\Lambda)^\mu \theta = \Lambda, \ldots, (\Lambda)^\mu \theta = \Lambda) : A \]

In previous slides, we referred to this type of abstraction as subset structured abstract domains.

Abstract domains are the abstract state variables and are the concrete variables.

Point-wise abstraction and Applications
Consider the program

Example: Program ANY-Y

\[
\begin{align*}
0 & = h = x \quad \text{Initially, integer } x, \\
\text{while } 0 & = x \\
\text{do} & \\
0 & = h = x
\end{align*}
\]
With the mapping \( \varphi \), we can obtain the abstract version of ANY-Y calls ANY-Y \( \varphi \).

\[
\begin{align*}
\{ \text{zero}, \text{pos}\} \supseteq \lambda \quad \square \quad : \varphi
\end{align*}
\]

The original invariance property \( \psi \), is abstracted into:

\[
\begin{align*}
\text{initially } \neg \text{zero} & \quad \text{neg } = \lambda \\
\text{while } 0 = X \text{ do } & \quad \text{endif}
\end{align*}
\]

\[
\begin{align*}
\begin{cases}
\text{pos} & \text{if } \neg \text{zero} \\
\text{neg} & \text{else}
\end{cases} =: \lambda : [1]
\end{align*}
\]

With the mapping \( \varphi \), we can obtain the abstract version of ANY-Y \( \varphi \), called ANY-Y \( \varphi \).

The Abstracted Version

A. Pnueli

Lecture 6: Predicate Abstraction and Applications

Program Analysis, NYU, Spring, 2004
Computing Abstractions of Assertions

Let $\mathcal{A}$ be an abstraction mapping and $\mathcal{I}$ be an assertion over the concrete variables. The $\alpha$-abstraction of can be computed by:

$$
\begin{align*}
(\mathcal{A})d & \vee (\mathcal{A})^\alpha = V^\mathcal{A} & \mathcal{A}E
\end{align*}
$$

For example

$$
(\mathcal{A})d \vee (\mathcal{A})^\alpha = V^\mathcal{A} \wedge \mathcal{A}E
$$

Computing Abstractions of Assertions
Let $\text{beanabstractionmappingand}$

\[
\text{pos} = \lambda \exists \neg \text{zero else} (I + \text{neg then } \lambda = \lambda \lor (\text{neg then } \lambda = \lambda \lor (\text{neg then } \lambda = \lambda)) : \text{neg then } \lambda \exists \neg \text{zero else} (I + \text{neg then } \lambda = \lambda \lor (\text{neg then } \lambda = \lambda) \lor \text{neg then } \lambda = \lambda) : \text{neg then } \lambda \exists \neg \text{zero else} (I + \text{neg then } \lambda = \lambda \lor (\text{neg then } \lambda = \lambda) \lor \text{neg then } \lambda = \lambda)
\]

For example:

\[
((\Lambda)^f = \lambda \lor (\Lambda)^{\text{pos}} = \lambda \lor (\Lambda)^{\text{neg}} = \lambda) : \lambda \lor (\Lambda)^{\text{pos}} = \lambda \lor (\Lambda)^{\text{neg}} = \lambda) : \lambda \lor (\Lambda)^{\text{pos}} = \lambda \lor (\Lambda)^{\text{neg}} = \lambda)
\]

The $d$-abstraction of $\Lambda$ can be computed by:

\[
((\Lambda)^f = \lambda : d)
\]

Let $\Lambda$ be an abstraction mapping and $((\Lambda)^{\text{pos}} = \lambda : \neg \Lambda)$ be an abstraction mapping and $((\Lambda)^{\text{neg}} = \lambda : \neg \Lambda)$ be an abstraction mapping and $((\Lambda)^{\text{pos}} = \lambda : \neg \Lambda)$ be an abstraction mapping and $((\Lambda)^{\text{neg}} = \lambda : \neg \Lambda)$.
Predicate Abstraction

The mapping is called a predicate abstraction \( \varphi \) if it contains the boolean equation \( \forall d = \exists d^r B, \ldots, \exists d^r B, \forall d^l, \exists d^l \) for each atomic state formula. The abstraction mapping \( \varphi \) is defined by \( \varphi \).

Let \( P \) be the set of all atomic formulas referring to the data (non-control) variables appearing within conditions in the program and within the temporal formula \( \phi \). Define abstract boolean variables \( C^r, D^r, C^l, D^l \) one for each atomic data formula appearing in \( P \) and within the abstract boolean variables. Following [BBM95] (and [GS97]), define abstract boolean variables \( C^r, D^r, C^l, D^l \), one for each atomic data formula. The abstraction mapping \( \varphi \) is called a predicate abstraction if it contains the boolean equation \( \forall d = \exists d^r B, \ldots, \exists d^r B, \forall d^l, \exists d^l \) for each atomic state formula occurring in \( \phi \), \( \Theta \), \( \Phi \), and \( \Psi \).

The mapping \( \varphi \) is called a predicate abstraction.
Example: Program BAKERY-2

The temporal properties for program BAKERY-2 are

\[
\begin{align*}
\Box & : acc_i \\
\neg \Box & : exc_i
\end{align*}
\]

Program Analysis, NYU, Spring, 2004
The abstracted properties can now be model-checked.

\[\begin{align*}
(0, 1) &= (\forall y > 1 y, B = 0, B) : \forall y \\
&\quad \text{Critical} \quad \forall y \\
&\quad \text{wait} \quad B = 0 \\
(0, 0) &= (\forall y > 1 y, B = 0, B) : \exists y \\
&\quad \text{Non-Critical} \quad \forall y \\
&\quad \text{loop forever do} \\
0 &= \forall y > 1 y, B = 0 = \forall y, B = 0 = 1 y, B \\
&\quad \text{where} \\
&\quad \text{local} B = 0, B = 0 = 1 y, B \\
&\quad \text{boolean} B = 0, B = 0 = 1 y, B \\
&\quad \text{and} \quad B = 0, B = 0 = 1 y, B
\end{align*}\]
Sometimes the initial predicate base is not sufficient. Consider the following example:

We wish to verify the invariance property

\[ \square ( \neg \phi \rightarrow \phi ) = 0. \]

\[
\begin{array}{c}
\text{while } 0 \neq x \\
\text{do}
\end{array}
\]

\[
\begin{array}{c}
I - x := x \\
x :=: \bar{h} \\
I + x :=: \bar{h}
\end{array}
\]

We are not able to refine the initial predicate base without additional assumptions.
A first attempt will make a sign abstraction of both $x$ and $y$, obtaining the following:

| $0^y \mathrel{\text{if}} 0^y, 0^x \mathrel{\text{else}} I, 0^{\varphi}j$ | $I, 0^y \mathrel{\text{if}} 0^y, 0^x \mathrel{\text{else}} I, 0^{\varphi}j$ | $I, 0^y \mathrel{\text{if}} 0^y, 0^x \mathrel{\text{else}} I$ | $0, 0^y \mathrel{\text{if}} 0^y, 0^x \mathrel{\text{else}} 0^y j$
|---|---|---|---|
| $\{I, 0\} \mathrel{\text{else}} I, \mathrel{\text{if}} 0 \mathrel{\text{then}} 0 \mathrel{\text{else}} I, \mathrel{\text{if}} 0 \mathrel{\text{then}} 0 \mathrel{\text{else}} I$ | $\{I, 0\} \mathrel{\text{else}} I, \mathrel{\text{if}} 0 \mathrel{\text{then}} 0 \mathrel{\text{else}} I$ | $\{I, 0\} \mathrel{\text{else}} I, \mathrel{\text{if}} 0 \mathrel{\text{then}} 0 \mathrel{\text{else}} I$ | $\{0, I, 0\} \mathrel{\text{else}} I, \mathrel{\text{if}} 0 \mathrel{\text{then}} 0 \mathrel{\text{else}} I$
| $X \mathrel{\text{:=}} I, 0^x j$ | $X \mathrel{\text{:=}} I, 0^x j$ | $X \mathrel{\text{:=}} I, 0^x j$ | $X \mathrel{\text{:=}} I, 0^x j$
| $\text{while} 0 \neq X \mathrel{\text{do}}$ | $\text{while} 0 \neq X \mathrel{\text{do}}$ | $\text{while} 0 \neq X \mathrel{\text{do}}$ | $\text{while} 0 \neq X \mathrel{\text{do}}$
| $I, 0 \rightarrow$ | $I, 0 \rightarrow$ | $I, 0 \rightarrow$ | $I, 0 \rightarrow$

This implies that we need to distinguish the concrete value $I$ from other positive concrete values. We find out that the last concrete step is inconsistent with the abstract step.

Concretizing this counter example, we obtain:

$$\langle I : \lambda, 0 : X \mathrel{\text{of}}, 0^y j \rangle, \langle I : \lambda, 0 : X \mathrel{\text{of}}, 0^y j \rangle, \langle I : \lambda, 0 : X \mathrel{\text{of}}, 0^y j \rangle, \langle I : \lambda, 0 : X \mathrel{\text{of}}, 0^y j \rangle$$

Trying to verify $Q \mathrel{\text{of}}$, we fail, producing the following counter example:

$$\langle I : \lambda, 0 : X \mathrel{\text{of}}, 0^y j \rangle, \langle I : \lambda, 0 : X \mathrel{\text{of}}, 0^y j \rangle, \langle I : \lambda, 0 : X \mathrel{\text{of}}, 0^y j \rangle, \langle I : \lambda, 0 : X \mathrel{\text{of}}, 0^y j \rangle$$

A first attempt will make a sign abstraction of both $x$ and $y$, obtaining the following:

First Attempt at Abstraction

Lehrenkind: Predicate Abstraction and Applications

A. Pnueli

Program Analysis, NYU, Spring 2004
This time, verification of \( \forall t \in [0, \infty) \) succeeds.

\[
\begin{align*}
\{2, 1\} \text{ if } 1 \text{ then } 0 \text{ else } 1 = \lambda if \quad 0 \text{ then } X \quad = \lambda if \quad 0 \text{ then } X \quad = X \quad : \mathbb{R} \\
\{2, 1\} \quad : \mathbb{R} \\
\{2, 1\} \quad \text{else } 0 \neq X \\
\text{while } 0 \neq X \quad : \mathbb{R} \\
\text{do } 0 \neq X \\
1 \quad : \mathbb{R} \\
0 \quad : \mathbb{R} \\
\{1\} \quad \text{else } 1 \neq I \quad : \mathbb{R} \\
\{0\} \quad \exists X \quad : \mathbb{R} \\
\{2\} \quad \text{else } 1 \neq I \quad : \mathbb{R} \\
\{0\} \quad \exists X \quad : \mathbb{R} \\
\end{align*}
\]
This abstracted program may diverge!

Consider the program LOOP.

Not all properties can be proven by pure finitary state abstraction.

Abstraction May Fail to Capture Some Properties
Application of Abstract Interpretation

Application of Abstract Interpretation

- Assertion checking and discovery.
- Program transformations and optimizations.
- Abstract testing and safety checking.

Taken from Lecture IV of David Schmidt.
Abstract testing and model generation - via model checking.

Once we generate an abstract model, we can analyze it further—ask

\{ B | s \}

is some \( S \in A \) such that \( \forall (s) \in A \).

\[ x = x \div 2; \]
\[ x = 4 \times x; \]
\[ \text{exit} \]

\( \forall (s) \in A \).

Form of abstract testing:

\[
\begin{array}{c}
\{ \ldots -2, 0, 2, \ldots \} = \{ \ldots -2, 0, 2, \ldots \}.
\end{array}
\]

Each trace tree denotes an abstract "test" that covers a set of concrete test cases, e.g., \( \forall (\text{even}) \).

Each trace tree denotes an abstract "test" that covers a set of concrete test cases, e.g., \( \forall (\text{even}) \).

\[ p_3, \text{even} \]
\[ p_2, \text{odd} \]
\[ p_0, \text{even} \]
\[ p_0, \text{odd} \]

\[ x = 3d \]
\[ x = 7d \]
\[ x = Id \]
\[ x = 0d \]

Abstract testing and model generation
Low-level safety checking

There are a variety of errors that might be checked on an abstract model; one example is type casting:

Perhaps a static analysis calculates the abstract store arriving at \( p_1 \):

\[
\begin{align*}
&\text{assuming } \forall (T) = \top. \\
&\text{because } \text{definite error, because } \text{Boolean,} \\
&\text{possible error — remove run-time check.} \\
&\text{no error possible — remove the run-time check.} \\
&\text{definite error, because } \text{non}. \\
\end{align*}
\]

There are a variety of errors that might be checked on an abstract model.

...
The approach to safety checking:

1. Design a Galois connection \( \mathcal{A} \subseteq \mathcal{C} \), such that all "checkpoint conditions," \( \mathcal{A} \subseteq \mathcal{C} \), are abstracted exactly by \( \mathcal{A} \subseteq \mathcal{C} \). Thus, no violation \( \mathcal{A} \subseteq \mathcal{C} \implies \mathcal{A} \subseteq \mathcal{C} \) implies \( \mathcal{A} \subseteq \mathcal{C} \). Therefore, no value/dead-code, then an error is possible.

When \( \mathcal{A} \subseteq \mathcal{C} \), then no error is possible; if \( \mathcal{A} \nsubseteq \mathcal{C} \), then an error is possible.

If \( \mathcal{A} \nsubseteq \mathcal{C} \), then no error is possible; if \( \mathcal{A} \nsubseteq \mathcal{C} \), then an error is possible.

2. For each checkpoint, \( \mathcal{A} \subseteq \mathcal{C} \), at program point \( \mathcal{P} \), for each abstract value \( \mathcal{A} \subseteq \mathcal{C} \), check if \( \mathcal{A} \subseteq \mathcal{C} \).

If \( \mathcal{A} \subseteq \mathcal{C} \), that arrives at \( \mathcal{P} \), check if \( \mathcal{A} \nsubseteq \mathcal{C} \).

have that \( \mathcal{A} \nsubseteq \mathcal{C} \) yet \( \mathcal{A} \subseteq \mathcal{C} \).

Otherwise, we might consider \( \mathcal{A} \nsubseteq \mathcal{C} \) or (equivalently, \( \mathcal{A} \nsubseteq \mathcal{C} \)).

1. Design a Galois connection \( \mathcal{A} \subseteq \mathcal{C} \), such that all "checkpoint conditions," \( \mathcal{A} \subseteq \mathcal{C} \), are abstracted exactly by \( \mathcal{A} \subseteq \mathcal{C} \). (That is, conditions, \( \mathcal{A} \subseteq \mathcal{C} \), are abstracted exactly by \( \mathcal{A} \subseteq \mathcal{C} \).)
from each \( p : \text{Error} \) \text{if store} \( T \), working backwards to see if an initial state is reached.

Analysis: 
\[
\text{interval analysis, where values have form, } \langle t \pm \epsilon, \langle \epsilon \pm \delta, \langle \delta \pm \gamma, \langle \eta \pm \zeta, \langle \zeta \pm \theta \rangle \rangle \rangle \rangle \]

Checkpoints:
for \( \epsilon \) \in \{K, T\} or \( \perp \).

Analyze: constant propagation, where values are either \( \epsilon \), \( \mu \), or \( \lambda \).

Uninitialized variables, dead-code, and erroneous-state checks:
\[
[1, 2] \cdot 1, 2, 3 - 1, 1 + 2, 3 \]

Checkpoints: for \( \eta \) \in [0, \alpha, \text{length}] \{ \}

Analysis: interval analysis, where values have form, \( [t, \epsilon] \), \( \epsilon \).

Array-bounds and arithmetic over- and under-flow checks:
Two more examples:
Program transformation: Constant folding

Basic principle of program transformation:

\[
\text{If } A \in S \text{ arrives at point } p^1, \text{ then } S \text{ can be replaced by } S' \text{ at } p^1.
\]

For constant folding, the transformation criteria are the abstract integers \(\ldots -1, 0, 1, \ldots\) (but not all). The analysis tells us to replace \(\gamma\) at point \(p^1\) by:

- \(\downarrow\) at point \(p^1\)
- \(\downarrow\) at point \(p^1\)
- \(\downarrow\) at point \(p^1\)
- \(\downarrow\) at point \(p^1\)
- \(\downarrow\) at point \(p^1\)

\[
\text{Const:}
\]

- \(\uparrow\) at point \(p^1\)
- \(\uparrow\) at point \(p^1\)
- \(\uparrow\) at point \(p^1\)
- \(\uparrow\) at point \(p^1\)
- \(\uparrow\) at point \(p^1\)

\[
\text{exnI : } 3d
\]

\[
\{ \text{exnI : } 3d \}
\]

\[
\text{Id : } 2d
\]

\[
(\exists d \in \mathbb{Z}) (x + k > x) \quad \text{whence}
\]

\[
\text{Const:}
\]

\[
\text{exnI : } 3d
\]

\[
\{ \text{exnI : } 3d \}
\]

\[
\text{Id : } 2d
\]

\[
(\exists d \in \mathbb{Z}) (x + k > x) \quad \text{whence}
\]
Program transformation: Code motion

A compiler translates a program into blocks of three-address code:

```plaintext
prod = 0;
i = 1;
do {
    prod = prod + a[i] * b[i];
i = i + 1;
} while i <= 20
```

The translation sometimes generates inefficient code, as array-indexing expressions are expanded:

```plaintext
L:
    prod = 0
    i = 1
    t1 = a[i]
    t2 = addr(a) - 4
    t3 = t2[t1]
    t4 = b[i]
    t5 = addr(b) - 4
    t6 = t5[t1]
    t7 = t3 * t6
    prod = prod + t7
    i = i + 1
    if i <= 20 goto L
```

Note: This example comes from the Aho and Ullman "green dragon" compiling text.

Ex: 
```plaintext
WHILE i <= 20 {
    t1 = i
    t1 = t1 + i
    prod = prod + a[t1] * b[t1] * prod
} do
    i = i + 1
    if i <= 20 goto L
```
A reaching-definitions analysis helps calculate that the statements in the loop’s body that assign to \( t_2 \) and \( t_4 \) are constant — the assignments can be moved out of the loop.
Condition checking and assertion synthesis

A backwards-necessarily analysis can synthesize assertions that ensure achievement of a postcondition.
The forwards-backwards analysis can be repeatedly alternated.

\[ pos = (\neg \text{notneg})^{1+} \# t o \neg \text{notneg} = (a)^{1+} \#
\]
\[ \bot = (t)_{0\neq}^{1+} f o r a \neq 0 \neq 1
\]
\[ \neg \text{notneg} = (\neg \text{notneg})^{0\neq} \# f o r a \neq 0 \neq 1
\]
\[ a o 0 = \neg f o r a \neq 0 \neq 1
\]

where

\[ \text{halt} : \text{x} \]
\[ I - x = x : \text{z d} \]
\[ I + x = x : \text{l d} \]
\[ 0 = x \quad \text{\#} \quad I 0 : \text{notneg : x} \]

The entry condition can be used with a forwards-possibly analysis to generate postconditions that sharpen the assertions.
General assertion checking

Checking user-supplied assertions is a form of low-level safety checking, e.g., we might check

\[ \neg (\exists x \in \mathbb{R} \forall y \in \mathbb{R} (x^2 + y^2 < 1)) \]

It is crucial that an assertion, \( \phi \in C \) (say, for \( C = \text{store} \)), be exactly checked, e.g., we might check

\[ \{ \text{assert } \phi \}{ } = 0 \]

Checcking user-supplied assertions is a form of low-level safety.

Nonetheless, the \textbf{Signs} domain defines its own "logic".

Let \( \phi \) abbreviate \{ \text{store} \} \forall \text{ holds for } s:

\[ x \in \{ \cdots -2, -1, 0, 1, 2, \cdots \} \]

\[ \text{Example: } x = 0, \text{ that is, } \]

\[ \{ \text{assert } \phi \} \]

is not exactly abstracted in \textbf{Signs}:

\[ \text{none, zero, neg, notpos, all} \]

Nonetheless, the \textbf{Signs} domain defines its own "logic".

\[ \forall x \left( \forall y \left( \neg (x^2 + y^2 < 1) \right) \right) \]

The underlying issue is \( \forall y \left( \neg (y^2 < 1) \right) \).