Program Transformation & Abstract Interpretation

In semantics-based (offline) program transformation, such as:

- constant propagation,
- partial evaluation,
- slicing,

abstract interpretation is classically used in a preliminary program static analysis phase:

- to collect the information about the program runtime behaviors,
- and determine which transformations are applicable.

---

Present Objective

Our present goal is quite different:

- Formalize the program transformation itself;
  With two objectives:
  - a program transformation correctness proof method;
  - a program transformation design methodology.

- Abstract interpretation is the appropriate framework to reach these objectives.
Abstract Interpretation

- **Abstract interpretation** formalizes the **conservative approximation** of the semantics of computer systems.

  Approximation: observation of the behavior of a computer system at some level of abstraction, ignoring irrelevant details;

  Conservative: the approximation cannot lead to any erroneous conclusion.

Abstract Interpretation (Cont’d)

- **Thinking tool**: the idea of abstraction by conservative approximation is central to reasoning (in particular on computer systems);

- **Mechanical tools**: the idea of effective approximation leads to automatic semantics-based program manipulation tools.

A Few Applications of Abstract Interpretation

Techniques involving approximations are naturally formalized by abstract interpretation:

- **Static Program Analysis** [POPL 77, 78, 79]
- **Hierarchies of Semantics (including Proofs)** [POPL 92, TCS 02]
- **Typing** [POPL 97]
- **Model Checking** [POPL 00]
- **Program Transformation** [POPL 02]
**Very Basic Elements of Abstract Interpretation Theory**

**Abstraction** \( \alpha \)

\[
\{ x : [1, 99], y : [2, 77] \}
\]

**Concretization** \( \gamma \)

\[
\{ x : [1, 99], y : [2, 77] \}
\]

\[
\begin{align*}
\text{The Abstraction } \alpha & \text{ is Monotone} \\
X \subseteq Y & \Rightarrow \alpha(X) \subseteq \alpha(Y)
\end{align*}
\]
The Concretization $\gamma$ is Monotone

$$X \subseteq Y \Rightarrow \gamma(X) \subseteq \gamma(Y)$$

The $\gamma \circ \alpha$ Composition

$$\{x : [1, 99], y : [2, 77]\} = \alpha \circ \gamma(Y) = Y$$

The $\alpha \circ \gamma$ Composition

$$\langle P, \subseteq \rangle \xleftarrow{\gamma} \xrightarrow{\alpha} \langle Q, \sqsubseteq \rangle$$

is defined as

- $\alpha$ is monotone
- $\gamma$ is monotone
- $X \subseteq \gamma \circ \alpha(X)$
- $\alpha \circ \gamma(Y) \sqsubseteq Y$

iff

$X \subseteq \gamma \circ \alpha(X)$

---

1 for short, more precisely “semi-dual Galois connections”.

2 see [POPL 79] for equivalent formalizations using closure operators, ideals, etc. and [JLC 92] for weaker hypotheses if no best approximation.
Function Abstraction

\[ F^\# = \alpha \circ F \circ \gamma \]

Abstract domain

\( F^\# \)

Concrete domain

\( F \)

\( \gamma \)

\( \alpha \)

\( \langle P, \subseteq \rangle \xrightarrow{\gamma} \langle Q, \subseteq \rangle \Rightarrow \langle P \xrightarrow{\text{mon}} P, \subseteq \rangle \xrightarrow{\lambda F \cdot \alpha \circ F \circ \gamma} \langle Q \xrightarrow{\text{mon}} Q, \subseteq \rangle \)

Approximate Fixpoint Abstraction

\( F^\# = \alpha \circ F \circ \gamma \Rightarrow \alpha(\text{lfp } F) \subseteq \text{lfp } F^\# \)

Online Program Transformation

(1) Online Program Transformation

- Program transformation is a syntactic process;
- maps a subject program into a transformed program;
- Both subject and transformed programs are syntactic objects.
(2) Online Program Transformation

- Program transformations refer to the semantics of the subject and transformed programs:
  - Online program transformations use values manipulated during program execution, hence directly refer to the source concrete semantics;
  - Offline program transformations use a preliminary static analysis of the source program, hence refer to its abstract semantics;

(3) Online Program Transformation

- The subject semantics and transformed semantics are different in general;
- However they should be equivalent, at some level of observation.
(3) Online Program Transformation

- The observational equivalence gets rids of irrelevant details about the subject and transformed program semantics;
- Hence it is an abstract interpretation of the subject and transformed program semantics!

(3) Example: Partial Evaluation

Subject program:
\[
\begin{align*}
Y & := 1 \\
X & := Y - 1
\end{align*}
\]

Transformed program:
\[
\begin{align*}
Y & := 1 \\
X & := 0
\end{align*}
\]

Subject program semantics:
\[
\begin{align*}
[X:0, Y:1] & \\
\downarrow Y & := 1 \\
[X:0, Y:1] & \\
\downarrow X & := Y - 1
\end{align*}
\]

Transformed program semantics:
\[
\begin{align*}
[X:0, Y:1] & \\
\downarrow Y & := 1 \\
[X:0, Y:1] & \\
\downarrow X & := 0
\end{align*}
\]

Observational abstraction:
\[
\begin{align*}
\alpha_o(S[P]) & = \\
\alpha_c(S[t[P]])
\end{align*}
\]

(4) Online Program Transformation

- The syntactic transformation induces a semantic transformation:
  The subject semantics is mapped to the transformed semantics;
- The subject semantics and the transformed semantics should be observationally equivalent;
- The semantic transformation is in general more precise than the algorithmic syntactic transformation (e.g. infinite behaviors).
(4) Online Program Transformation

Subject program $P$  
\[ \xrightarrow{\text{Syntactic transformation } t} \]  
Transformed program $t[P]$  

Subject program semantics $S[P]$  
\[ \xrightarrow{\text{Semantic transformation } t} \]  
Transformed program semantics $t[S[P]] \subseteq S[t[P]]$

$\alpha_O(S[P]) \leq \alpha_O(t[S[P]]) \leq \alpha_O(S[t[P]])$

(5) Correspondence Between Syntax and Semantics, Cont’d

- The correspondence between syntax and semantics is an abstraction:
  \[ \text{po} \langle \emptyset; \subseteq \rangle \xrightarrow{S} \text{po} \langle P/\#; \subseteq \rangle \]

- The concretization $S$ is the semantics of the program;
- The abstraction $p$ is the “decompilation” of the semantics.

(5) Correspondence Between Syntax and Semantics

- The program syntax forgets details about the program execution semantics:
  - The sequence of values of variables during execution is forgotten, but:
    - their existence and maybe their type are recorded;
    - the sequence (partial order, ...) of (denotations of) actions performed on these variables is recorded;
  - Program execution times are completely abstracted (but might be included in the operational semantics);
(6) Semantic Transformations as Approximations

- A **semantic program transformation** is a **loss of information** on the semantics of the subject program;
  —→ The **semantic program transformation** is an abstraction;

(6) Example: Partial Evaluation

\[
\begin{align*}
S[&Y := 1; \\
&X := 0;]
\end{align*}
\]

\[
\begin{align*}
S[&Y := 1; \\
&X := Y - 1;]
\end{align*}
\]

\[
\begin{align*}
S[&Y := 1; \\
&X := 2 \times Y - 2;]
\end{align*}
\]

\[
\begin{align*}
S[&Y := 1; \\
&X := Y \times (Y - 1);]
\end{align*}
\]

(7) Syntactic Transformations as Approximations

- By composition, the **syntactic program transformation** is also a **loss of information** on subject program;
  —→ The **syntactic program transformation** is an abstraction;
**Formalization of Program Transformation Correctness by Abstract Interpretation**

**Design of Program Transformations by Abstract Interpretation**
Design of an Online Program Transformation

Subject program $P$ $\xrightarrow{\gamma}$ Transformed program $t[P] \nsubseteq p[t[S[P]]]$

Subject program semantics $S[P]$ $\xrightarrow{\gamma}$ Transformed program semantics $t[S[P]] \subseteq S[t[P]]$

Semantic transformation $t$ $\vdash \leftarrow$ Observational abstraction

Transformed program $t[P] \nsubseteq p[t[S[P]]]$ $\xrightarrow{\gamma}$ Transformed program semantics $t[S[P]] \subseteq S[t[P]]$

Design of Program Transformation Algorithms

$t[P] \nsubseteq p[t[S[P]]]
= p[t[lfp \subseteq F[P]]]
= \ldots$

← apply fixpoint transfer/ approximation theorems (with widening)

We obtain an iterative program transformation algorithm; This algorithm is classical or new!
Principle of Offline Program Transformation

Program Transformations Formalized in the Paper

- Constant propagation;
- Online & offline partial evaluation;
- Mixline partial evaluation (with widening);
- Static program monitoring \( S[t[P, M]] = S[P] \cap S[M] \):
  - Example 1: run-time checks elimination,
  - Example 2: security,
  - Example 3: proof by transformation (\( P \otimes t[P, M] \)).

Illustrative Examples

Conclusion
Conclusion

• Program transformation is formalized as an abstraction of a semantic transformation of run-time execution;

• Leads to a unified framework for semantics-based program analysis and transformation;

• The benefit is presently purely foundational and conceptual;

• Pave the way to:
  – machine-checked program transformations,
  – a formalization of compilation.