Ludwig
Maximilians
Universität
München

# Seminararbeit

## Abstrakte Interpretation I

Pöcking, 10. Juli 2009

Ludwig-Maximilians-Universität München
Institut für Informatik
Lehr- und Forschungseinheit Theoretische Informatik
Öttingenstraße 67
D-80538 München

**Verfasser:** Nadezda Mikhaylova

**Dozent:** Prof. Martin Hofmann, Ph.D.
**Betreuer:** Roland Axelsson

# Inhaltsverzeichnis

# 1 Einleitung

Abstrakte Interpretation ist ein Konzept, das die Objekte eines Programms mit Hilfe von abstrakten Domänen interpretiert. Diese Theorie wurde schon 1977 von Patrick und Radhia Cousot vorgestellt. Die abstrakte Interpretation hat viele Anwendungsgebiete, aber besonders häufig wird sie zur Programmanalyse benutzt. Sie erlaubt eine statische Programmanalyse, mit der die Aussagen über ein Programm und seine Zustände gemacht werden können.

Ziel dieser Arbeit ist es, die grundlegenden Ideen der abstrakten Interpretation vorzustellen, ihre Eingenschaften, Grundgedanken und Anwendungsgebiete.

Im ersten Teil werden die theoretischen Grundlagen und Prinzipien der abstrakten Interpretation dargestellt. Es werden die Anwendungsgebiete und das Vorgehensweise anhand von Beispielen gezeigt. Der zweite Teil beschäftigt sich mit der Approximation der Fixpunkte, was eine der Techniken der abstrakten Interpretation ist. Dafür werden zunächst die mathematischen Grundlagen erklärt, auf denen diese Technik aufgebaut ist. Danach werden Widening und Narrowing Operatoren betrachtet, die bei der Approximation benutzt werden.

# 2 Grundlagen der abstrakten Interpretation

Abstrakte Interpretation - ist eine allgemeine Theorie für die semantische Approximation von diskreten dynamischen Systemen [PC05]. Ziel der abstrakten Interpretation ist es, Informationen über das Verhalten von Programmen zu bekommen, indem man Teile des Programms abstrahiert und die Anweisungen Schritt für Schritt nachvollzieht.

In der Pprogrammanalyse wird die abstrakte Interpretation für diverse Analyseverfahren eingesetzt:

- Model-Checking - Technik zum Beweisen der Gültigkeit einer Spezifikation in Bezug auf ein Programm. Gegeben sei eine Spezifikationsformel $\varphi$ (üblicherweise in einer Temporallogik) und ein Programm $P$. Model Checking beantwortet die Frage: Wird $\varphi$ von $P$ erfüllt? Dabei werden mehrere Zustände des Programms zu abstrakten Mengen zusammengefürt;

- Approximation der Fixpunkte - Annäherung an die Fixpunkte durch Iteration (wird im Teil 2 behandelt);

- Software Steganographie - abstrakte Informationen bei der Verschlüsselung von Daten. Die Information wird im Code so verschlüsselt, dass man sie nur durch die abstrakte Interpretation der konkreten Semantik des Codes entschlüsseln kann [PC04] ;

- WCET-Analyse - statische Festlegung der worst-case Zeit für den Ablauf eines Programms (oder eines seiner Teile, z.B Schleife), die das Analysieren des Zugriffsspeichers, des Ablaufs des Programms beeinflusst.

Klassische Anwendungen sind außerdemb Intervallgrenzen, Codeerreichbarkeit, Korrektheitsbeweise, Konstantenpropagation (die Untersuchung darauf, welche Variablenwerte sicher zur Compilezeit berechnet werden können), Mode-Analyse und andere [Pre00].

Bei diesen und anderen Methoden der Programmanalyse gibt es oft Schwierigkeiten, da die Programme häufig nicht vollständig ausgetestet werden können, weil es unendliche viele Eingabewerte (z.B. alle ganzen Zahlen) gibt und auch die Variablen Belegungen aus einem unendlich großen Werteraum haben können. Aus diesem Grund konzentriert man sich bei der abstrakten Interpretation auf die Teilaspekte der Ausführung der Anweisungen, man lässt

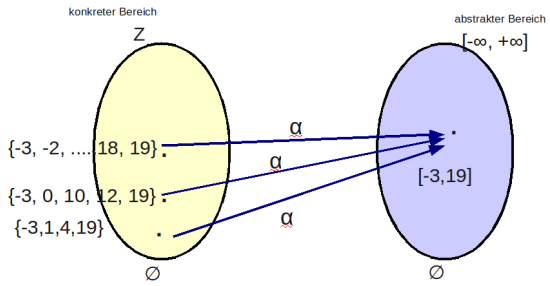


Abbildung 1: Abstraktionsfunktion alpha

einiges an Information weg und erhält letztlich eine Näherung an die Programmsemantik. Durch solche Abstraktionen werden unterschiedliche Programmpfade zusammengeführt, so dass die möglichen Werte der Variablen, abhängig von den abstrakten Wertebereichen, approximiert werden. Es gibt verschiedene Techniken, um die Daten zu abstrahieren. Bei den Galois Verbindungen definiert man beispielsweise zwei Funktionen:

- Abstraktion $\alpha$, die alle Werte auf seinen abstrakten Wert abbildet:
$$\alpha : \Sigma_\gamma \rightarrow \Sigma_\alpha$$

- Konkretisierung $\gamma$, die die einem abstrakten Wert alle konkrete Werte zuordnet, für die er steht:
$$\gamma : \Sigma_\alpha \rightarrow \Sigma_\gamma$$

Nach der Verwendung der Abstraktionsfunktion liefert die Konkretisierungsfunktion $\gamma$ einen Wert (eine Menge), aus der man einige konkrete Werte nicht rauslesen kann, z.B
Wir haben drei Mengen {-3, 1, 7, 19 }, {-3, 0, 10, 12, 19} und {-3, -1, -1, ..., 18, 19} und die Funktion $\alpha(X) = [\min(x), \max(x)]$.

Alle drei Mengen werden mit dieser Funktion $\alpha$ auf den gleichen Intervall [-3, 19] abgebildet (Abb. 1).
Nach der Verwendung der Konkretisierung-Funktion $\gamma$ bekommt man die Menge {-3, ...., 19} (Abb. 2). Dieser Bereich bildet alle drei ursprünglichen Mengen ab, bei der ersten und der zweiten Menge sind aber die Werte nicht mehr rekonstruierbar. Diese Information geht verloren. Für die dritte Menge ist das Ergebnis zufällig frei von Informationsverlust.

Eine Möglichkeit ganze Zahlen zu abstrahieren wäre alle Zahlen aus Z durch even/odd zu ersetzen, je nachdem, ob die Zahl gerade oder ungerade

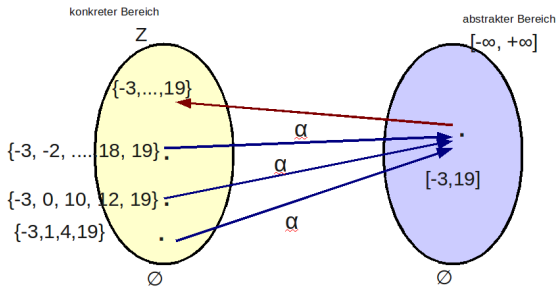


Abbildung 2: Konkretisierungsfunktion gamma

ist. Die abstrakte Funktion $\alpha$ wäre dann: $\alpha : Z \rightarrow \{even, odd\}$

$$\alpha(x) = \begin{cases} even & , \ falls\ x\ gerade \\ odd & , \ falls\ x\ ungerade \end{cases} \tag{1}$$

Eine andere Möglichkeit wäre, die ganzen Zahlen in Äquivalenzklassen zu unterteilen, je nachdem ob die Zahl kleiner, gleich oder größer 0 ist
$\{+, 0, -\}$:
$\alpha : Z \rightarrow \{+, 0, -\}$, oder

$$\alpha(x) = \begin{cases} \{0\} & , \ falls\ x = 0 \\ \{+\} & , \ falls\ x > 0 \\ \{-\} & , \ falls\ x < 0 \end{cases} \tag{2}$$

Diese Möglichkeiten erlauben es Aussagen zu machen, ob die Werte eines Ausdrucks (einer Berechnung, eines Programmstücks usw.) gerade, ungerade bzw. größer, gleich oder kleiner 0 sind, ohne den Ausdruck selbst auszuwerten.

Ein weiteres einfaches Beispiel für die Verwendung der abstrakten Interpretation ist ein Verfahren, mit dem man die Korrektheit von arithmetischen Berechnungen austesten kann. Beispielsweise wollen wir testen, ob die Berechnung

$$373 * 8847 + 12345 \overset{?}{=} 3312266$$

korrekt ist. Da wir diese Zahlen nicht so schnell berechnen können, versuchen wir es, von konkreten Werten zu abstrakten zu gehen. Wir nutzen dazu die Tatsache aus, dass eine Zahl durch 9 teilbar ist, genau dann wenn ihre Quersumme durch 9 teilbar ist. Z.B gilt:
$[(10 * a + b)\ mod\ 9] = [(a + b)\ mod\ 9]$ oder
$[a * b\ mod\ 9] = [((a\ mod\ 9) * (b\ mod\ 9))\ mod\ 9]$ und

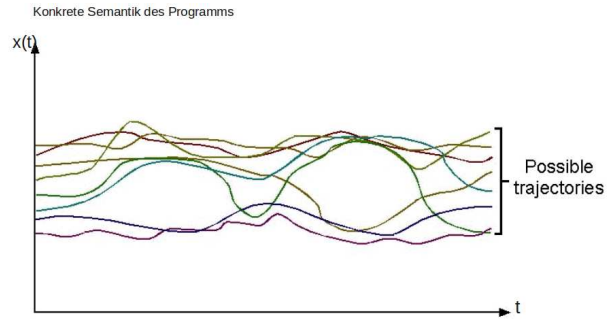


Abbildung 3: Konkrete Semantik eines Programms

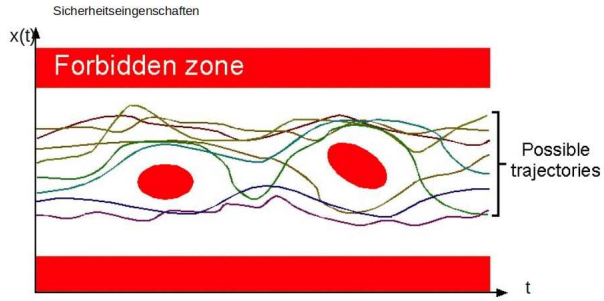


Abbildung 4: Sichere Pfade eines Programms

$[a + b \ mod \ 9] = [((a \ mod \ 9) + (b \ mod \ 9)) \ mod \ 9]$.
Mit diesem Wissen definieren wir die abstrakte Menge $M = \{0, ..., 8\}$, die Abstraktionsfunktion $\alpha$ und die Operatoren $\oplus$ und $\otimes$ auf $M$:
$\alpha(x) = x \ mod \ 9$
$\alpha_1 \oplus \alpha_2 = (\alpha_1 + \alpha_2) \ mod \ 9$
$\alpha_1 \otimes \alpha_2 = (\alpha_1 * \alpha_2) \ mod \ 9$
Es muss gelten:
$l_1 \times l_2 + l_3 = l_4 \Rightarrow \alpha(l_1) \otimes \alpha(l_2) \oplus \alpha(l_3) = \alpha(l_4)$
Wir bilden jetzt die iterierten Quersummen der vier obigen Zahlen und erhalten dabei 4 (für 373), 0 (für 8847), 6 (für 12345) und 5 (für 3312274). Wir führen nun die Berechnung auf den Quersummen durch und erhalten
$4 * 0 + 6 = 6$. Wir haben also auf beiden Seiten nicht dieselbe Quersumme bekommen, was aber der Fall sein müsste, wenn die Berechnung korrekt wäre [Koe05]. Wir wissen daher, dass obige Gleichung nicht gilt, ohne die ganze Berechnung durchzuführen. Es gilt aber nicht im Allgemeinen, wenn die Quersummen auf beiden Seiten gleich sind, dann ist die Berechnung korrekt, z.B: 111 * 2 = 6.

In der Programmanalyse kann man die abstrakte Interpretation intuitiv so ausdrücken: durch die abstrakte Semantik wird ein Gleichungssystem aufge-

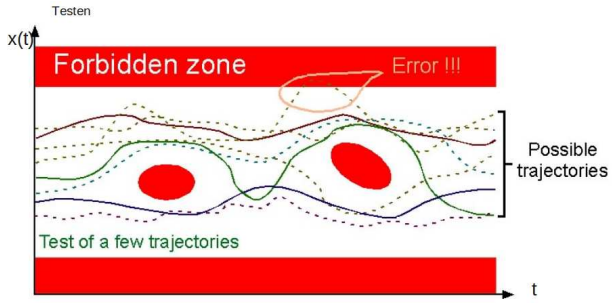


Abbildung 5: Testen

baut, dass für jeden Programmpunkt alle mögliche Zustände des Programms in allen möglichen Umgebungen berechnet. Was man dabei berücksichtigen soll, wird im folgenden betrachtet.

Um eine abstrakte Semantik zu bilden, muss man die konkrete Semantik des Programms kennen. Die konkrete Semantik formalisiert alle möglichen Abläufe des Programms in allen möglichen Umgebungen. Der Ablauf wird als mathematisches Modell repräsentiert durch die Kurven, die die Evolution des Vektors x(t) als Werte der Eingabe-, Zustands- und Ausgabevariablen als Funktion in der Zeit t repräsentieren (Abb. 3).

Die konkrete Semantik hat aber das Problem der Unentscheidbarkeit - sie ist im Allgemeinen unberechenbar als ein unendliches mathematisches Objekt. D. h. es ist unmöglich, ein Programm zu schreiben, das alle mögliche Abläufe und Zustände eines anderen Programms in allen seinen möglichen Umgebungen repräsentiert und berechnet. Deswegen sind viele nicht triviale Fragen bezüglich der konkreten Semantik des Programms unentscheidbar.

Spezifikation der Sicherheitseigenschaften – sichere Pfade eines Programm repräsentieren alle Abläufe, die sich mit der Umgebung den fehlerhaften (unzulässigen) Zuständen nicht überkreuzen. Grafisch werden diese fehlerhaften Zustände als Verbotene Zone (Forbidden zone") dargestellt (Abb. 4).

Prüfen der sicheren Pfade - die Verifikation der sicheren Pfade besteht in der Überprüfung, ob die Pfade der konkreten Semantik sich mit den unzulässigen Zonen nicht überkreuzen, was am häufigsten durch das Testen geschieht.

Testen (Debugging) besteht im Berücksichtigen einer Teilmenge aller möglichen Abläufe. Das Testen garantiert aber nicht die hundertprozentige Sicherheit, d.h. wenn sogar alle durchgeführten Tests fehlerfrei gelaufen sind, kann es passieren, dass einige fehlerhafte Fälle nicht überprüft wurden (Abb. 5). Damit alle Fälle abgedeckt werden, verwendet man die abstrakte Interpretation.

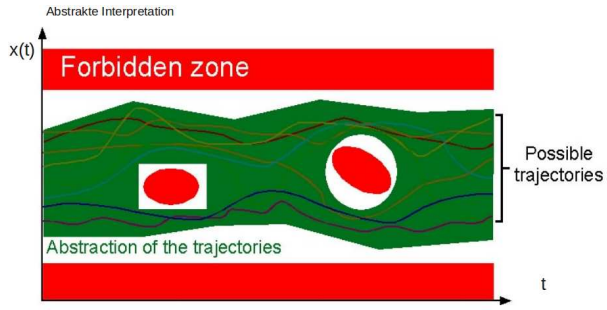


Abbildung 6: Abstrakte Semantik

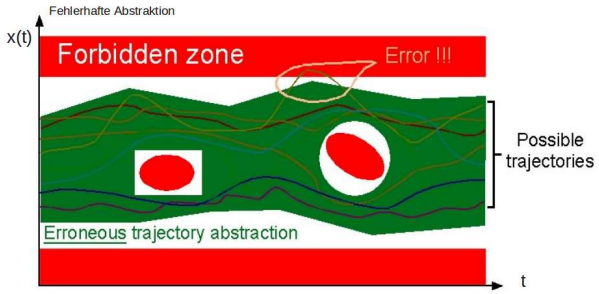


Abbildung 7: Fehlerhafte Interpretation

Abstrakte Interpretation besteht in der Berücksichtigung der abstrakten Semantik, die man durch die Überapproximation der konkreten Programmsemantik bekommt. Diese abstrakte Semantik soll alle möglichen Werte abdecken. Man sagt, wenn die abstrakte Semantik sicher ist, dann ist auch die konkrete Semantik sicher (Abb. 6) [PC04].

Abbildungen 7 und 8 zeigen zwei Beispiele der Approximation der konkreten Semantik. Im ersten Beispiel nimmt ein Pfad einen unzulässigen Wert an, der aber nicht in der abstrakten Semantik berücksichtigt war. In diesem Fall haben wir eine fehlerhafte Abstraktion – die abstrakte Semantik muss immer als Überapproximation der konkreten Semantik gewählt werden, sonst können die abstrakten Werte nicht fehlerfrei auf konkrete Werte übertragen werden. Im zweiten Beispiel verursacht die Überapproximation ein Fehler. In diesem Fall prüft man, ob dieser Wert in der konkreten Semantik vorkommen kann. Wenn nicht, war das ein falscher Alarm, man sollte die abstrakte Semantik entsprechend anpassen.

Nach diesen theoretischen Ansätzen beschäftigen wir uns mit einem der Anwendungsgebiete der abstrakten Interpretation der Approximation der Fixpunkte.

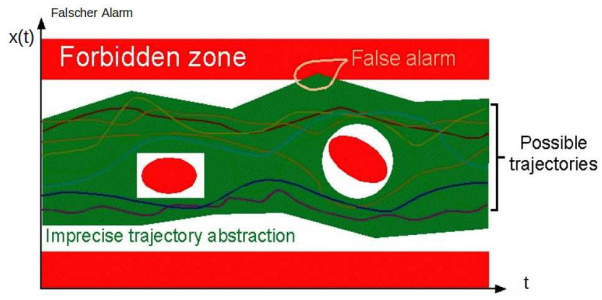


Abbildung 8: Falscher Alarm

# 3 Approximation der Fixpunkte

## 3.1 Mathematische Grundbegriffe

Die Approximation der Fixpunkte ist eine Technik der Programmanalyse, die auf mathematischen Theorien aufbaut und somit mathematisch bewiesen werden kann. Vor allem der Begriff des vollständigen Verbandes spielt dabei eine große Rolle. Deshalb müssen fürs Verständnis dieses Verfahrens erst einige mathematischen Begriffe erklärt werden.

**Partielle Ordnung** $\sqsubseteq$ auf einer Menge M ist eine Relation, für die gilt:

- $\sqsubseteq$ ist reflexiv: $m \sqsubseteq m \; für \; jede \; m \in M$;
- $\sqsubseteq$ ist transitiv: aus $m_1 \sqsubseteq m_2 \; und \; m_2 \; \sqsubseteq \; m_3 \; folgt \; m_1 \; \sqsubseteq \; m_3, für \; m_1, \; m_2, \; m_3 \; \in \; M$;
- $\sqsubseteq$ ist antisymmetrisch: aus $m_1 \sqsubseteq m_2 \; und \; m_2 \; \sqsubseteq m_1 \; folgt \; m_1 = m_2$;

**Vollständiger Verband (complete lattice).** Ein Tupel (L, $\sqsubseteq$ ), bestehend aus einer Menge L und einer partiellen Ordnung auf L heißt vollständiger Verband, wenn jede Teilmenge Y von L eine kleinste obere (Supremum) und eine größte untere (Infimum) Schranke hat. Es muss insbesondere für Y = $\{\emptyset\}$ gelten. Man definiert $\top = sup \; L$ (top) und $\bot = inf \; L$ (bottom) [Koe05].

Beispiel:
$int_1 \sqsubseteq int_2 \; iff \; inf(int_2) \leq inf(int_1) \wedge sup(int_1) \leq sup(int_2)$ und
$(f^n(\bot)) \sqsubseteq \bigsqcup_n f^n(\bot)$

**Fixpunkt-Theorie.** Sei $f : L \rightarrow L$ eine monotone Funktion auf dem vollständigen Verband $L = (L, \sqsubseteq, \top, \bot)$. Ein Fixpunkt von f ist ein Element l $\in$ L, so dass f(l) = l. Die Menge aller Fixpunkte:

$$Fix(f) = \{l \in L \mid f(l) = l\}$$

Die Menge aller Präfixpunkte:

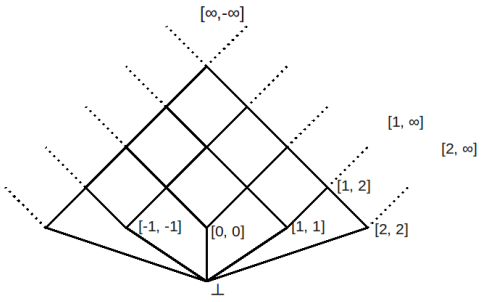


Abbildung 9: Vollständiger Verband

$$Pre(f) = \{l \in L \mid f(l) \sqsubseteq l\}$$

Die Menge aller Postfixpunkte:

$$Post(f) = \{l \in L \mid f(l) \sqsupseteq l\}$$

Der kleinste Fixpunkt (least fixed point):

$$lfp(f) = \bigsqcap Pre(f) \in Fix(f) \subseteq Pre(f)$$

Der größte Fixpunkt (greatest fixed point):

$$gfp(f) = \bigsqcup Post(f) \in Fix(f) \subseteq Post(f)$$

Theorem vom Knarski-Tarski
Sei (L, $\sqsubseteq$ ) ein vollständiger Verband und $f : L \to L$ eine monotone Funktion. Dann gilt:

$$lfp(f) = \bigsqcap Pre(f) \in Fix(f)$$
$$gfp(f) = \bigsqcup Post(f) \in Fix(f)$$

Folgerungen:

- jede monotone Funktion besitzt mindestens einen Fixpunkt;

- die Menge aller Fixpunkte bildet einen vollständigen Verband.

Fixpinkt-Iteration. Sei f eine monotone Funktion auf L, nach dem Satz von Kleene gilt:
wenn für jede aufsteigende Kette $(l_n)_n$ gilt: $f(\bigsqcup_{n=0}^{\infty} l_n) = \bigsqcup_{n=0}^{\infty} f(l_n)$, so gilt

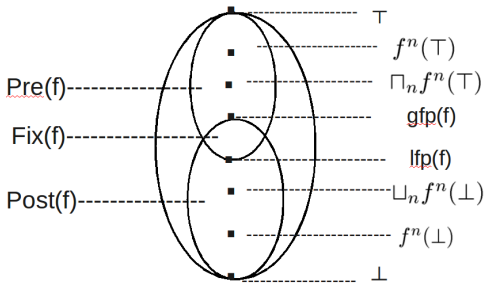


Abbildung 10: Fixpunkt-Iteration

$$lfp(f) = \bigsqcup_{n=0}^{\infty} f^n)(\bot) = f^m(\bot)$$

man berechnet also $f(\bot) \bigsqcup f(f(\bot)) \bigsqcup, ..., \bigsqcup f^i(\bot), ...$ bis die Folge stationär ist und erhält dann den kleinsten Fixpunkt. In der Abbildung 10 wird diese Iteration illustriert, man sieht, dass [FN05]:

$$(f^n(\bot)) \sqsubseteq \bigsqcup_n f^n(\bot) \sqsubseteq lfp(f) \sqsubseteq gfp(f) \sqsubseteq \bigsqcap_n f^n(\top) \sqsubseteq f^n(\top)$$

**Upper Bound Operator** Operator $\biguplus : \mathrm{L} \times \mathrm{L} \to \mathrm{L}$ auf dem vollständigen Verband L = (L,$\sqsubseteq$) heißt Upper Bound Operator, wenn:

$$l_1 \sqsubseteq (l_1 \biguplus l_) \sqsupseteq l_2 \text{ für alle l1, l2} \in \mathrm{L}$$

Es wird ein Element zurück geliefert, das immer größer ist, als beide Argumente.

## 3.2 Widening Operator

Bei der Fixpunkt-Iteration können einige Probleme auftreten, z.B es kann nicht garantiert werden, dass die Iteration terminiert, oder die Analyse kann abhängig vom Programm sehr zeitaufwendig sein (Schleife mit hoher Iterationszahl). Deshalb verwendet man häufig die Widening und Narrowing Operatoren, um die Annäherung an die Fixpunkte zu beschleunigen und in einigen Fällen überhaupt zu ermöglichen.[PC04]
Erst betrachten wir den Widening Operator.
Ein Operator $\nabla : L \times L \to L$ auf dem vollständigen Verband ist ein Widening Operator nur dann, wenn:

- Das ist ein Upper Bound Operator, und

- für alle aufsteigenden Ketten $(l_n)_n$ wird die aufsteigende Kette $(l_n^{\nabla})_n$ letztendlich stabil.

Widening Operator auf einem Intervall könnte man beispielsweise so definieren:
$[l_0,\ u_0] \nabla [l_1,\ u_1] = [\ if\ l_1 \leq l_0\ then\ -\infty\ else\ \ l_0;$
$if\ u_1 \geq u_0\ then + \infty\ else\ u_0]$

$[0,1] \nabla [0,2] = [0, +\infty]$
$[0,2] \nabla [0,2] = [0,2]$
Man sieht, dass der Widening Operator nicht monoton ist.
Mit dem Widening Operator und einer monotonen Funktion f : L × L → L kann man die neue Sequenz $(f_\nabla^n)_n$ kalkulieren:

$$(3) \qquad (f_\nabla^n) = \begin{cases} \bot & ,\ falls\ n = 0 \\ (f_\nabla^{n-1}) & ,\ falls\ n\ \geq\ 0 \wedge f(f_\nabla^{n-1}) \sqsubseteq (f_\nabla^{n-1}) \\ (f_\nabla^{n-1}) \nabla f(f_\nabla^{n-1}) & ,sonst \end{cases}$$

Falls die Iterationsanzahl n der Sequenz gleich 0 ist, startet die Berechnung mit dem Wert ⊥ für alle Variablen in allen Zuständen. Für das Fall, dass bereits Iterationen stattgefunden haben und die erneute Anwendung der Funktion f den gleichen Element (Intervall oder Teilmenge) liefert, so ist ein Fixpunkt erreicht und die Menge bleibt bestehen, sonst wird der Widening Operator auf dem letzten und dem neu berechneten Wert angewendet (dritter Fall) [Cor08] .

Die neue Sequenz ist eine aufsteigende Kette, die letztendlich stabil wird. Außerdem sehen wir aus der Folgerung (ii) (unten), dass wir für einige m $f(f_\nabla^m) \sqsubseteq (f_\nabla^m)$ haben. Das bedeutet, dass f reduzierend ist und aus der Knarski-Tarski Theorem wissen wir, dass $f_\nabla^m \sqsupseteq lfp(f)$ gelten muss. Man schreibt:

$$lfp_\nabla(f) = f_\nabla^m$$

Der Widening-Operator wird also angewendet, bis die zurückgeliferten Werte gleich sind, was garantiert, dass der kleinste Fixpunkt in der Menge enthalten ist (Überapproximation).

Folgerungen. Wenn ∇ ein Widening Operator ist, dann gilt:

i. Sequenz $(f_\nabla^n)$ ist eine aufsteigende Kette;

ii. wenn $f(f_\nabla^m) \sqsubseteq f_\nabla^m$ für einige m, dann wird die Sequenz $(f_\nabla^n)_n$ letztendlich stabil und für alle $n \geq m : f_\nabla^n = f_\nabla^m\ und\ \bigsqcup_n f_\nabla^n = f_\nabla^m$;

iii. wenn $(f_\nabla^n)_n$ stabil ist, dann existiert ein m, so dass $f(f_\nabla^m) \sqsubseteq (f_\nabla^m)$, und

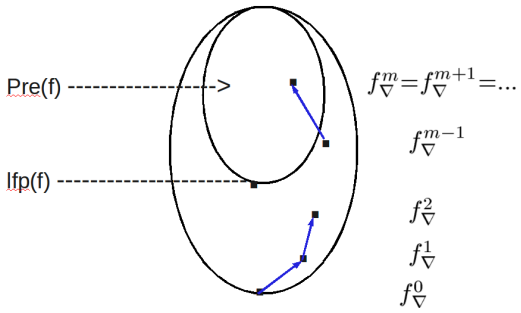


Abbildung 11: Widening Operator

iv. $\bigsqcup_n f_\nabla^n \sqsupseteq lfp(f)$.

Beispiel [PC04]. Wir haben eine Schleife und vier Programmzählestände der Variable x (1 bis 4): bei der Initialisierung, in der Schleifenbedingung, nach der Berechnung in der Schleife und nach dem Beenden der Schleife. Die Gleichung rechts definiert alle Werte vom x. Wir wollen sicher stellen, dass kein Overflow auftitt. Am Anfang sind alle Zustände gleich $\emptyset$ (nicht definiert):

| | |
|---|---|
| (1) | |
| x := 1; | $X_1 = [1,1]$ |
| 1: | $X_2 = (X_1 \sqcup X_3) \cap [-\infty, 9999]$ |
| while x $\leq$ 10000 do | $X_3 = X_2 \oplus [1,1]$ |
| 2: | $X_4 = (X_1 \sqcup X_3) \cap [10000, \infty]$ |
| x := x+1 | |
| 3: | |
| od; | $X_1 = \emptyset$ |
| 4: | $X_2 = \emptyset$ |
| | $X_3 = \emptyset$ |
| | $X_4 = \emptyset$ |

(2) Im ersten Schritt initialisieren wir $X_1$:

| | |
|---|---|
| x := 1; | $X_1 = [1,1]$ |
| 1: | $X_2 = (X_1 \sqcup X_3) \cap [-\infty, 9999]$ |
| while x $\leq$ 10000 do | $X_3 = X_2 \oplus [1,1]$ |
| 2: | $X_4 = (X_1 \sqcup X_3) \cap [10000, \infty]$ |
| x := x+1 | |
| 3: | |
| od; | $X_1 = [1,1]$ |
| 4: | $X_2 = \emptyset$ |
| | $X_3 = \emptyset$ |

$$X_4 = \emptyset$$

(3) In den nächsten zwei Schritte berechnen wir die Werte von $X_2$ und $X_3$

| | |
|---|---|
| x := 1; | $X_1 = [1,1]$ |
| 1: | $X_2 = (X_1 \sqcup X_3) \cap [-\infty, 9999]$ |
| while x $\leq$ 10000 do | $X_3 = X_2 \oplus [1,1]$ |
| 2: | $X_4 = (X_1 \sqcup X_3) \cap [10000, \infty]$ |
| x := x+1 | |
| 3: | |
| od; | $X_1 = [1,1]$ |
| 4: | $X_2 = [1,1]$ |
| | $X_3 = [2,2]$ |
| | $X_4 = \emptyset$ |

(4) Der Wert von $X_1$ bleibt unverändert, der Wertebereich von $X_2$ und $X_3$ wird um eins in jedem Schleifendurchlauf vergrößert:

| | |
|---|---|
| x := 1; | $X_1 = [1,1]$ |
| 1: | $X_2 = (X_1 \sqcup X_3) \cap [-\infty, 9999]$ |
| while x $\leq$ 10000 do | $X_3 = X_2 \oplus [1,1]$ |
| 2: | $X_4 = (X_1 \sqcup X_3) \cap [10000, \infty]$ |
| x := x+1 | |
| 3: | |
| od; | $X_1 = [1,1]$ |
| 4: | $X_2 = [1,2]$ |
| | $X_3 = [2,2]$ |
| | $X_4 = \emptyset$ |

(5) Nach einigen Schritten erreicht $X_3$ den Wert [2,6], da die Schleife aber bis 10000 läuft, verwenden wir den Widening-Operator, um den Prozess zu beschleunigen:

| | |
|---|---|
| x := 1; | $X_1 = [1,1]$ |
| 1: | $X_2 = (X_1 \sqcup X_3) \cap [-\infty, 9999]$ |
| while x $\leq$ 10000 do | $X_3 = X_2 \oplus [1,1]$ |
| 2: | $X_4 = (X_1 \sqcup X_3) \cap [10000, \infty]$ |
| x := x+1 | |
| 3: | |
| od; | $X_1 = [1,1]$ |
| 4: | $X_2 = [1,2]$ |
| | $X_3 = [2,6]$ |
| | $X_4 = \emptyset$ |

(6)

| | |
|---|---|
| x := 1; | $X_1 = [1,1]$ |
| 1: | $X_2 = (X_1 \sqcup X_3) \cap [-\infty, 9999]$ |
| while x $\leq$ 10000 do | $X_3 = X_2 \oplus [1,1]$ |
| 2: | $X_4 = (X_1 \sqcup X_3) \cap [10000, \infty]$ |
| x := x+1 | |
| 3: | |
| od; | $X_1 = [1,1]$ |
| 4: | $X_2 = [1, +\infty]$ |
| | $X_3 = [2,6]$ |
| | $X_4 = \emptyset$ |

(7) Aus den neuen Wert von $X_2$ bekommen wir neuen $X_3$

| | |
|---|---|
| x := 1; | $X_1 = [1,1]$ |
| 1: | $X_2 = (X_1 \sqcup X_3) \cap [-\infty, 9999]$ |
| while x $\leq$ 10000 do | $X_3 = X_2 \oplus [1,1]$ |
| 2: | $X_4 = (X_1 \sqcup X_3) \cap [10000, \infty]$ |
| x := x+1 | |
| 3: | |
| od; | $X_1 = [1,1]$ |
| 4: | $X_2 = [1, +\infty]$ |
| | $X_3 = [2, +\infty]$ |
| | $X_4 = \emptyset$ |

(8)

| | |
|---|---|
| x := 1; | $X_1 = [1,1]$ |
| 1: | $X_2 = (X_1 \sqcup X_3) \cap [-\infty, 9999]$ |
| while x $\leq$ 10000 do | $X_3 = X_2 \oplus [1,1]$ |
| 2: | $X_4 = (X_1 \sqcup X_3) \cap [10000, \infty]$ |
| x := x+1 | |
| 3: | |
| od; | $X_1 = [1,1]$ |
| 4: | $X_2 = [1, 9999]$ |
| | $X_3 = [2, +\infty]$ |
| | $X_4 = \emptyset$ |

(9) $X_3$ erreicht den Wert von 10000, die Schleife wird beendet und wir können im nächsten Schritt den Wert von $X_4$ berechnen

| | |
|---|---|
| x := 1; | $X_1 = [1,1]$ |
| 1: | $X_2 = (X_1 \sqcup X_3) \cap [-\infty, 9999]$ |
| while x $\leq$ 10000 do | $X_3 = X_2 \oplus [1,1]$ |

| | |
|---|---|
| 2: | $X_4 = (X_1 \sqcup X_3) \cap [10000, \infty]$ |
| x := x+1 | |
| 3: | |
| od; | $X_1 = [1, 1]$ |
| 4: | $X_2 = [1, 9999]$ |
| | $X_3 = [2, +10000]$ |
| | $X_4 = \emptyset$ |

(10)

| | |
|---|---|
| x := 1; | $X_1 = [1, 1]$ |
| 1: | $X_2 = (X_1 \sqcup X_3) \cap [-\infty, 9999]$ |
| while x $\leq$ 10000 do | $X_3 = X_2 \oplus [1, 1]$ |
| 2: | $X_4 = (X_1 \sqcup X_3) \cap [10000, \infty]$ |
| x := x+1 | |
| 3: | |
| od; | $X_1 = [1, 1]$ |
| 4: | $X_2 = [1, 9999]$ |
| | $X_3 = [2, +10000]$ |
| | $X_4 = [+10000, +10000]$ |

(11) Am Ende schreiben wir alle vier Werte von x in den entsprechenden Zeilen

| | |
|---|---|
| x := 1; | $X_1 = [1, 1]$ |
| 1: $\{x = 1\}$ | $X_2 = (X_1 \sqcup X_3) \cap [-\infty, 9999]$ |
| while x $\leq$ 10000 do | $X_3 = X_2 \oplus [1, 1]$ |
| 2: $\{x \in [1, 9999]\}$ | $X_4 = (X_1 \sqcup X_3) \cap [10000, \infty]$ |
| x := x+1 | |
| 3: $\{x \in [2, 10000]\}$ | |
| od; | |
| 4: $\{[x = 10000\}$ | |

Mit Hilfe des Widening-Operators haben wir nach den wenigen Schritten das Ende der While-Schleife erreicht und haben gesehen, dass wir keinen Overflow haben, da x einen endlichen Wert angenommen hat.

Da der Widening-Operator häufig zu unendlich springt, bekommt man die Lösung, die allerdings ziemlich grob ist. Aus diesem Grund verwendet man den Widening Operator zusammen mit dem Narrowing Operator, der den Wertebereich wieder einschränkt.

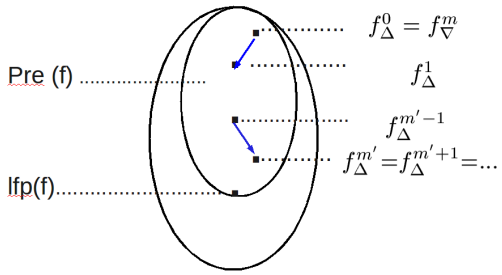


Abbildung 12: Narrowing Operator

## 3.3 Narrowing Operator

Ein Operator $\Delta : L \times L \rightarrow L$ auf dem vollständigen Verband L ist ein Narrowing Operator nur dann, wenn

- $l_2 \sqsubseteq l_1 \Rightarrow l_2 \sqsubseteq (l_1 \Delta l_2) \sqsubseteq l_1 f\ddot{u}r\ alle\ l_1, l_2 \in L$, und
- für alle absteigenden Ketten $(l_n)_n$ wird die Sequenz $(l^\Delta_n)_n$ letztendlich stabil.

Da $l(l^m_\nabla)$ eine absteigende Kette ist, können wir die neue Sequenz $(f^n_\Delta)_n$ berechnen:

$$(f^n_\Delta) = \begin{cases} f^m_\nabla & , \ falls\ n = 0 \\ f^{n-1}_\Delta \ \Delta \ f(f^{n-1}_\Delta) & , \ falls\ n \ \geq \ 0 \end{cases} \tag{4}$$

Folgerung 1. Wenn $\Delta$ Narrowing Operator ist und $f(f^m_\nabla) \sqsubseteq f^m_\nabla$, dann

- $(f^n_\Delta)_n$ eine absteigende Kette in der Menge Pre(x), und
- $f^n_\Delta \sqsupseteq f^n(f^m_\nabla) \sqsupseteq \ lfp(f)\ f\ddot{u}r\ alle\ n.$

Beweis. Wir wollen zeigen, dass

$$f^{n+1}(f^m_\nabla) \sqsubseteq f(f^n_\Delta) \sqsubseteq f^{n+1}_\Delta \sqsubseteq f^n_\Delta$$

Wir zeigen das durch die vollständige Induktion.
Für n=0 mit $f(f^m_\nabla) \sqsubseteq f^m_\nabla$ folgt sofort, dass

$$f^{n+1}(f^m_\nabla) \sqsubseteq f(f^n_\Delta) \sqsubseteq f^n_\Delta$$

Bei der Konstruktion $f^{n+1}_\Delta$ nehmen wir $f(f^n_\Delta) \sqsubseteq f^{n+1}_\Delta \sqsubseteq f^n_\Delta$. Die beiden

sind unsere Voraussetzung der Induktion.
Induktionsschritt:

$$f^{n+2}(f_\nabla^m) \sqsubseteq f^2(f_\Delta^n) \sqsubseteq f(f_\Delta^{n+1}) \sqsubseteq f(f_\Delta^n)$$

Nach der Induktionsvoraussetzung haben wir $f(f_\nabla^n) \sqsubseteq f_\Delta^{n+1}$ und daraus folgt:

$$f^{n+2}(f_\nabla^m) \sqsubseteq f(f_\Delta^{n+1}) \sqsubseteq f_\Delta^{n+1}$$

Bei der Konstruktion $f_\nabla^{n+2}$ erhalten wir:

$$f(f_\Delta^{n+1}) \sqsubseteq f_\Delta^{n+2} \sqsubseteq f_\Delta^{n+1}$$

Ende der Induktion. Aus der Induktion folgt, dass $(f_\Delta^n)_n$ eine absteigende Kette in Pre(f) ist $\Rightarrow f^n(f_\nabla^m) \sqsubseteq f_\Delta^n \; für \; n \geq 0$. Und aus der Annahme $f(f_\nabla^m) \sqsubseteq f_\nabla^m$ folgt sofort, dass

$$f(f^n(f_\nabla^m)] \sqsubseteq f^n(f_\nabla^m) für \; n \geq 0 \text{ und } f^n(f_\nabla^m) \in Pre(f).$$

Und daraus folgt, dass $f^n(f_\nabla^m) \sqsupseteq lfp(f)$. Ende des Beweises.

Folgerung 2. Wenn $\Delta$ Narrowing Operator ist und $f(f_\nabla^m) \sqsubseteq f_\nabla^m$, dann wird die absteigende Kette $(f_\Delta^n)_n$ letztendlich stabil.
Beweis. Wir definieren zuerst die Sequenz $(l_n)_n$ mit:

$$l_n = \begin{cases} f_\nabla^m & , \; falls \; n = 0 \\ f(f_\Delta^{n-1}) & , \; falls \; n \geq 0 \end{cases} \tag{5}$$

Diese Sequenz ist eine absteigende Kette, weil $(f_\Delta^n)_n$ eine ist und weil $f(f_\Delta^0) \sqsubseteq f_\nabla^m$. Jetzt prüfen wir durch die Induktion, dass $l_n^\Delta = f_\Delta^n$.
Für n = 0 folgt das sofort. Für den Induktionsschritt kalkulieren wir

$$l_{n+1}^\Delta = l_n^\Delta \Delta l_{n+1} = f_\Delta^n \Delta f(f_\Delta^n) = f_\Delta^{n+1}$$

Daraus folgt, dass $(f_\Delta^n)_n$ letztendlich stabil wird.

Folgerung 1 garantiert, dass die Sequenz $f_\Delta^n$ (aus (4), S.16) eine absteigende Kette ist, deren alle Elemente die Bedingung $lfp(f) \sqsubseteq f_\Delta^n$ erfüllen. Folgerung 2 sagt uns, dass diese Kette letztendlich stabil wird, so dass $f_\Delta^{m'} = f_\Delta^{m'+1}$ für beliebige m'. Deswegen können wir schreiben

$$lfp_\nabla^\Delta(f) = f_\Delta^{m'}$$

als die Approximation an lfp(f) (Abb. 11 und 12).

Beispiel. Der vollständige Verband aus der Abbildung 9 hat zwei Typen der unendlich absteigenden Ketten: eine enthält die Elemente der Form $[-\infty, z]$, die zweite der Form $[z, \infty]$, *wo* $z \in Z$. Wir nehmen die Kette der zweiten Form, eine unedlich absteigende Kette mit den Elementen:

$$[z_1, \infty], [z_2, \infty], [z_3, \infty], ...$$

wo $z_1 \leq z_2 \leq z_3 \leq ....$ Wir wollen einen Narrowing-Operator $\Delta_N$ definieren, der die Sequenz beim $z_i \geq N$ (N ist ein positiver Integer) stabilisiert. Wir definieren $\Delta = \Delta_N$:

$$(6) \qquad int_1 \Delta int_2 = \begin{cases} \bot & , \; falls \; int_1 = \bot \vee int_2 = \bot \\ [z_1, z_2] & , sonst \end{cases}$$

wo gilt:

$$(7) \qquad z_1 = \begin{cases} inf(int_1) & , \; falls \; N \leq inf(int_2) \wedge sup(int_2) = \infty \\ inf(int_2) & , sonst \end{cases}$$

$$(8) \qquad z_2 = \begin{cases} sup(int_1) & , \; falls \; inf(int_2) = -\infty \wedge sup(int_2) \leq -N \\ sup(int_2) & , sonst \end{cases}$$

Wir betrachten eine unendlich absteigende Kette $[n, \infty]_n$:

$$[0, \infty], [1, \infty], [2, \infty], [3, \infty], [4, \infty], [5, \infty]...$$

und eine N = 3. Dann liefert der oben definierte Operator $\Delta_N$ die Sequenz $([n, \infty]^{\Delta})_n$:

$$[0, \infty], [1, \infty], [2, \infty], [3, \infty], [3, \infty], [3, \infty], ...$$

[FN05].

# 4 Zusammenfassung

Abstrakte Interpretation ist semantik-basiert und damit unabhängig von konkreten Notationen und Sprachen. Viele ihrer Techniken (Galois Verbindungen, Widening und Narrowing Operatoren und andere) werden häufig in der Programmanalyse verwendet. Mit Hilfe der abstrakten Interpretation kann man z.B das Korrektheit eines Programmausdruckes überprüfen, die Werte, welche die Programmvariablen annehmen dürfen festlegen oder Terminierung des Programms bestimmen.
Es wurden außerdem die Widening und Narrowing Operatoren betrachtet, welche zur Beschleunigung der Approximation der Fixpunkte verwendet werden.

# Abbildungsverzeichnis

# Literatur


[Cor08] Daniel Cordes. *Schleifenanalyse fuer einen WCET-optimierenden Compiler basierend auf Abstrakter Interpretation und Polylib.* Dortmund, 2008. S.31-51.

[FN05] Chris Hankin Flemming Nielson, Hanne Riis Nielson. *Principles of Program Analysis.* Springer-Verlag, Berlin, 2005. S.211-245.

[Koe05] Barbara Koenig. *Automatische Analyse und Verifikation von Programmen (Skript zur Vorlesung im Sommersemester 2005 an der Universitaet Stuttgart).* Stuttgart, 2005. S.35-76.

[PC04] Radhila Cousot Patrick Cousot. *An Abstract Interpretation-Based Framework for Software Watermarking*, 2004. http://cristal.inria.fr/POPL2004/Abstracts/57.html.

[PC05] Jerome Hunsaker Patrick Cousot. *An Informal Overview of Abstract Interpretation*, 2005. http://web.mit.edu/afs/athena.mit.edu/course/16/16.399/www/.

[Pre00] Alexander Pretschner. *Abstarkte Interpretation.* Muenchen, 2000.