

Verlässliche Echtzeitsysteme

Grundlagen der statischen Programmanalyse

Peter Ulbrich

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)

www4.informatik.uni-erlangen.de

05. Mai 2015



- Wie **verifiziert** man die Eigenschaften (funktional/nicht-funktional) von Software stichhaltig?
 - Dynamische Codeanalyse (Testen) (s. Kapitel IV) **meist unzureichend!**
 - **Korrektheit im Allgemeinen nicht berechenbar/entscheidbar!**



Analyse und Vereinfachung der Programmsemantik

- **Statische Codeanalyse:**
automatische Extraktion von Programmeigenschaften
- **Abstrakte Interpretation:**
Analysemethodik unter Zuhilfenahme von Approximation



- **Korrektheitsaussagen** sind schwer zu formulieren!
 - Auch wenn nur eine **bestimmten Programmeigenschaft** relevant ist!
 - Wie hilft uns „**Abstrakte Interpretation**“ bei diesem Problem?
- Was sind die **mathematischen Grundlagen** abstrakter Interpretation?
 - Eine „informelle“ Sichtweise auf die Zusammenhänge
- **Ziel:** **Grobes Verständnis** abstrakter Interpretation entwickeln!



1 Überblick

2 Problemstellung

- Beispiel
- Konkrete Programmsemantik

3 Abstrakte Interpretation

- Abstrakte Semantik
- Sammelsemantiken
- Präfixsemantiken

4 Mathematische Grundlagen (Selbststudium)

5 Zusammenfassung



```
1 unsigned int average(unsigned int *array,  
2                     unsigned int size)  
3 {  
4     unsigned int temp = 0;  
5  
6     for(unsigned int i = 0; i < size; i++) {  
7         temp += array[i];  
8     }  
9  
10    return temp/size;  
11 }
```

■ Wo könnte es hier klemmen?

- Ist der Zugriff auf Feld array in Zeile 7 korrekt?
- Kann die Addition in Zeile 7 überlaufen?
- Kann in Zeile 10 eine Division durch 0 auftreten?



```
1 unsigned int average(unsigned int *array,  
2                     unsigned int size)  
3 {  
4     unsigned int temp = 0;  
5  
6     for(unsigned int i = 0; i < size; i++) {  
7         temp += array[i];  
8     }  
9  
10    return temp/size;  
11 }
```

■ Wo könnte es hier klemmen?

- Ist der Zugriff auf Feld array in Zeile 7 korrekt?
- Kann die Addition in Zeile 7 überlaufen?
- Kann in Zeile 10 eine Division durch 0 auftreten?



Fehlersuche: Was kann hier alles schief gehen?


Die Gretchenfrage der Softwareentwicklung ...

```
1 unsigned int average(unsigned int *array,  
2                     unsigned int size)  
3 {  
4     unsigned int temp = 0;  
5  
6     for(unsigned int i = 0; i < size; i++) {  
7         temp += array[i];  
8     }  
9  
10    return temp/size;  
11 }
```

■ Wo könnte es hier klemmen?

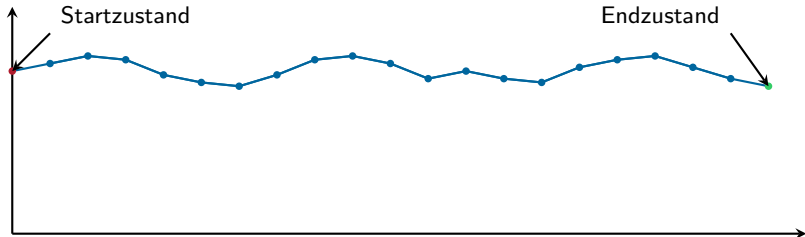
- Ist der Zugriff auf Feld array in Zeile 7 korrekt?
- Kann die Addition in Zeile 7 überlaufen?
- Kann in Zeile 10 eine Division durch 0 auftreten?

■ Wie findet man das heraus?

 Schauen wir mal, wie sich das Programm verhält.



Das Verhalten zur Laufzeit ist entscheidend!



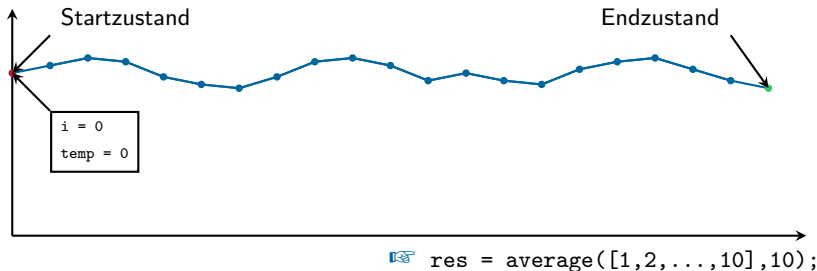
```
res = average([1,2,...,10],10);
```

```
1 unsigned int average(uint *array,
2                       uint size)
3 {
4     uint temp = 0;
5
6     for(uint i = 0; i < size; i++) {
7         temp += array[i];
8     }
9
10    return temp/size;
11 }
```

i	temp
---	------



Das Verhalten zur Laufzeit ist entscheidend!

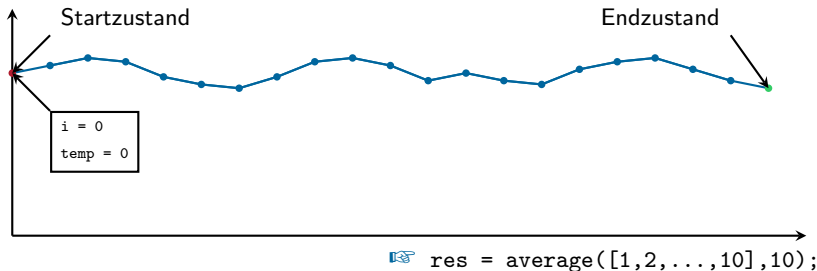


```
1 unsigned int average(uint *array,  
2                     uint size)  
3 {  
4     uint temp = 0;  
5  
6     for(uint i = 0; i < size; i++) {  
7         temp += array[i];  
8     }  
9  
10    return temp/size;  
11 }
```

i	temp
0	0



Das Verhalten zur Laufzeit ist entscheidend!

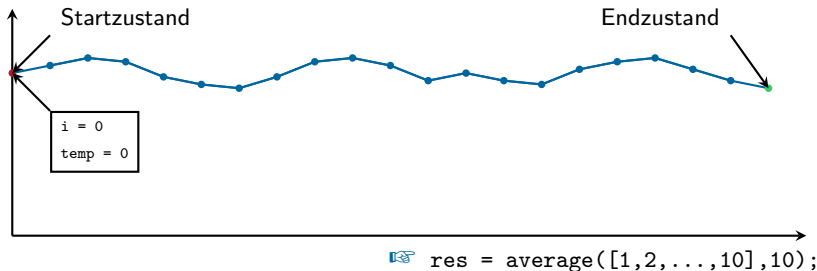


```
1 unsigned int average(uint *array,  
2                     uint size)  
3 {  
4     uint temp = 0;  
5  
6     for(uint i = 0; i < size; i++) {  
7         temp += array[i];  
8     }  
9  
10    return temp/size;  
11 }
```

i	temp
0	0
1	1



Das Verhalten zur Laufzeit ist entscheidend!

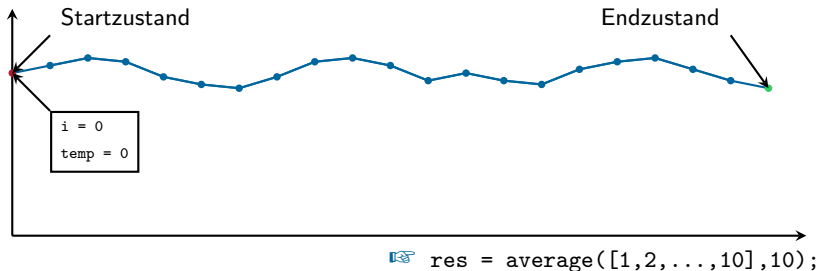


```
1 unsigned int average(uint *array,  
2                     uint size)  
3 {  
4     uint temp = 0;  
5  
6     for(uint i = 0; i < size; i++) {  
7         temp += array[i];  
8     }  
9  
10    return temp/size;  
11 }
```

i	temp
0	0
1	1
2	3



Das Verhalten zur Laufzeit ist entscheidend!

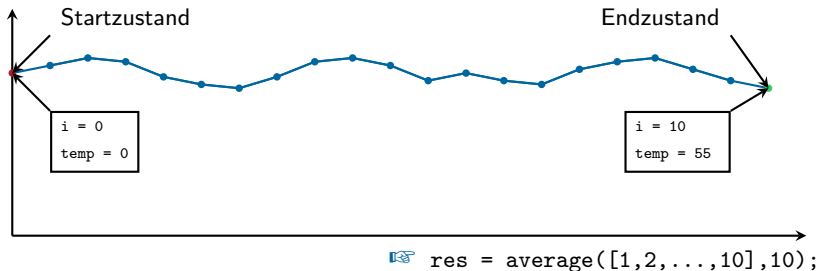


```
1 unsigned int average(uint *array,  
2                     uint size)  
3 {  
4     uint temp = 0;  
5  
6     for(uint i = 0; i < size; i++) {  
7         temp += array[i];  
8     }  
9  
10    return temp/size;  
11 }
```

i	temp
0	0
1	1
2	3
3	6



Das Verhalten zur Laufzeit ist entscheidend!



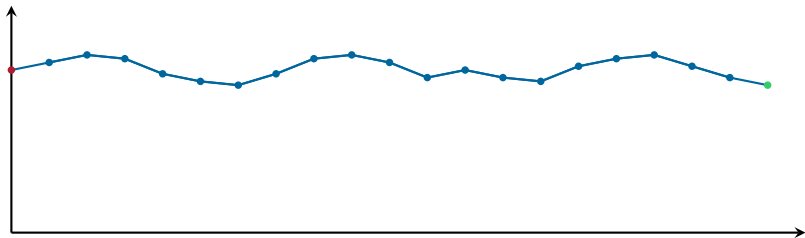
```
1 unsigned int average(uint *array,  
2                     uint size)  
3 {  
4     uint temp = 0;  
5  
6     for(uint i = 0; i < size; i++) {  
7         temp += array[i];  
8     }  
9  
10    return temp/size;  
11 }
```

i	temp
0	0
1	1
2	3
3	6
...	...
10	55



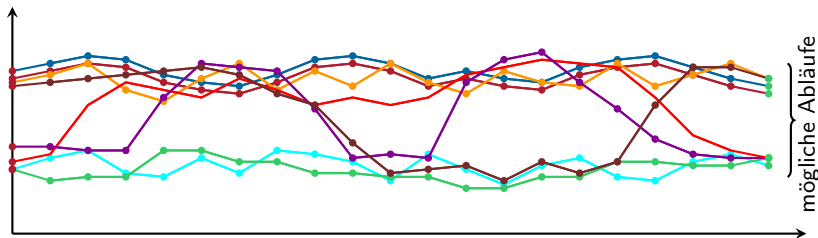
Konkrete Programmsemantik

Eine informelle Einführung in die Prinzipien abstrakter Interpretation [2]

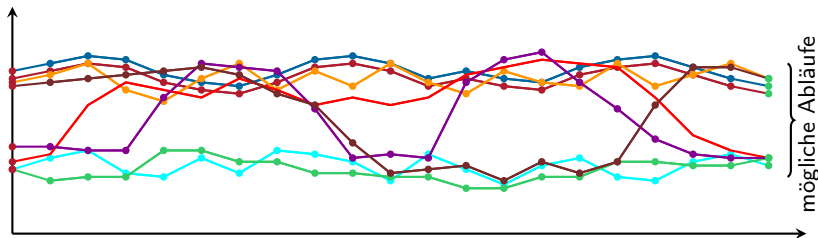


- Die **konkrete Semantik** (engl. *concrete semantics*) beschreibt
 - Alle möglichen Ausführungen eines Programms
 - Unter allen möglichen Ausführungsbedingungen



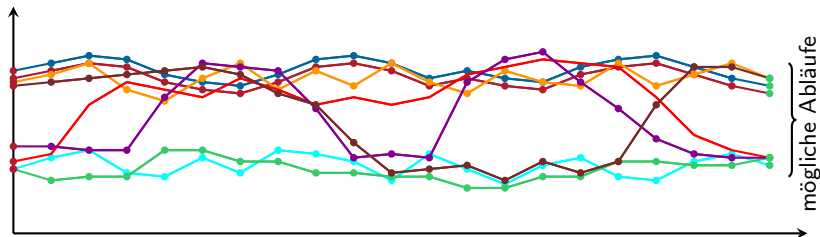


- Die **konkrete Semantik** (engl. *concrete semantics*) beschreibt
 - Alle möglichen Ausführungen eines Programms
 - Unter allen möglichen Ausführungsbedingungen
 - Für unser Beispiel bedeutet dies:
 - 2^{32} verschieden große Felder, 2^{32} verschiedene Werte für jedes Element

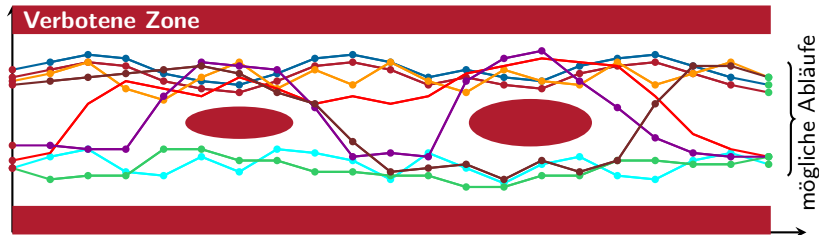


- Die **konkrete Semantik** (engl. *concrete semantics*) beschreibt
 - Alle möglichen Ausführungen eines Programms
 - Unter allen möglichen Ausführungsbedingungen
 - Für unser Beispiel bedeutet dies:
 - 2^{32} verschieden große Felder, 2^{32} verschiedene Werte für jedes Element
- Sie beschreibt ein „unendliches“ mathematisches Objekt
 - Im Allgemeinen **nicht berechenbar** durch einen Algorithmus
 - Alle nicht-trivialen Fragestellungen sind **nicht entscheidbar**

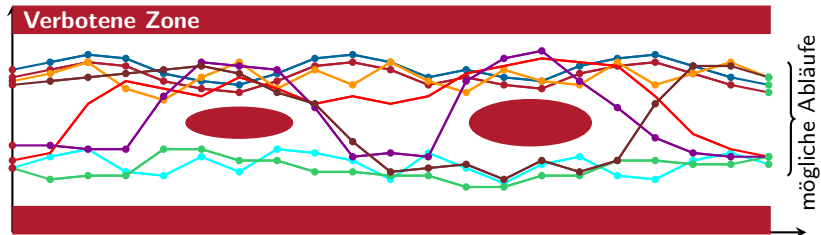




- Sicherheitseigenschaften (engl. *safety properties*) stellen sicher, dass keine fehlerhaften/unerwünschten Zustände eingenommen werden

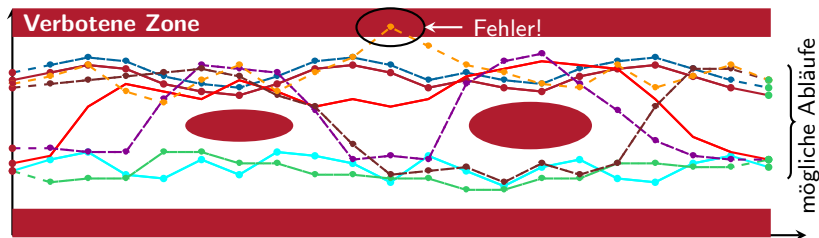


- Sicherheitseigenschaften (engl. *safety properties*) stellen sicher, dass keine fehlerhaften/unerwünschten Zustände eingenommen werden
- Ein Sicherheitsnachweis (engl. *safety proof*) garantiert, dass die konkrete Semantik nie eine verbotene Zone durchläuft



- Sicherheitseigenschaften (engl. *safety properties*) stellen sicher, dass keine fehlerhaften/unerwünschten Zustände eingenommen werden
 - Ein Sicherheitsnachweis (engl. *safety proof*) garantiert, dass die konkrete Semantik nie eine verbotene Zone durchläuft
- Das ist ein unentscheidbares Problem
- Die konkrete Programmsemantik ist nicht berechenbar

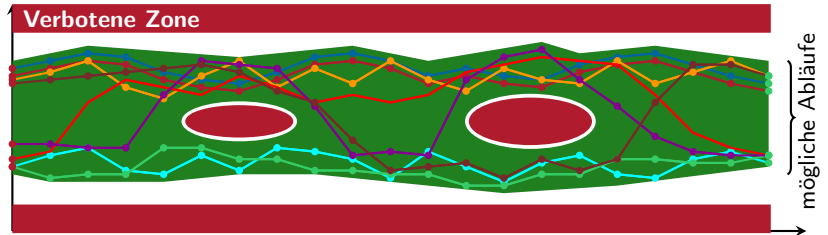




- Testen betrachtet **nur eine Teilmenge** aller möglichen Ausführungen
 - ↪ Gut geeignet, um die **Existenz** von Defekten zu zeigen
 - ↪ Ungeeignet, um ihre **Abwesenheit** zu zeigen
 - Evtl. hat man die fehlerhafte Ausführung einfach nicht getestet
- Problem: **unzureichende Abdeckung** der konkreten Semantik

- 1 Überblick
- 2 Problemstellung
 - Beispiel
 - Konkrete Programmsemantik
- 3 Abstrakte Interpretation
 - Abstrakte Semantik
 - Sammelsemantiken
 - Präfixsemantiken
- 4 Mathematische Grundlagen (Selbststudium)
- 5 Zusammenfassung





- Abstrakte Interpretation (engl. *abstract interpretation*)
 - Betrachtet eine *abstrakte Semantik* (engl. *abstract semantics*)
 - Sie umfasst **alle Fälle der konkreten Programmsemantik**
 - Ist die abstrakte Semantik sicher \Rightarrow konkrete Semantik ist sicher

Formale Methoden sind abstrakte Interpretationen

Die abstrakte Semantik wird aber auf unterschiedliche Weise bestimmt

Model Checking

- Abstrakte Semantik wird explizit vom Nutzer angegeben
→ endliche Beschreibung der konkreten Programmsemantik
 - Z.B. endliche Automaten, Aussagen- oder Prädikatenlogik
- Automatische Ableitung durch **statische Analyse**

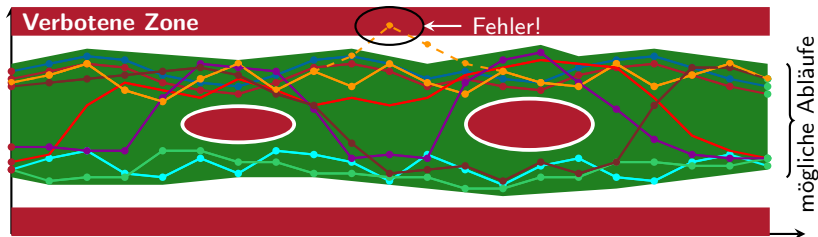
Deduktive Methoden

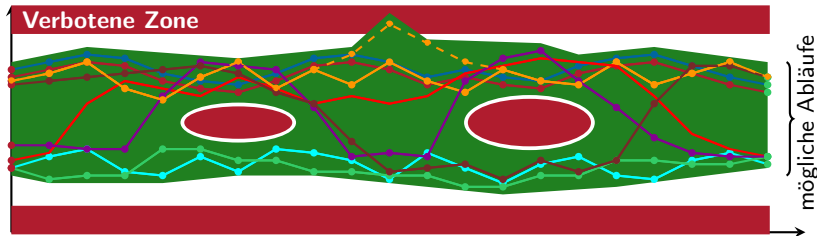
- Abstrakte Semantik wird durch Nachbedingungen beschrieben
- Nutzer gibt sie durch induktive Argumente an
 - Z.B. Vorbedingungen und Invarianten
- Automatische Ableitung durch **statische Analyse**

Statische Analyse

- Abstrakte Semantik wird ausgehend vom Quelltext bestimmt
 - Abbildung auf **vorab bestimmte, wohldefinierte Abstraktionen**
- Anpassungen (automatisch/durch den Nutzer) sind möglich

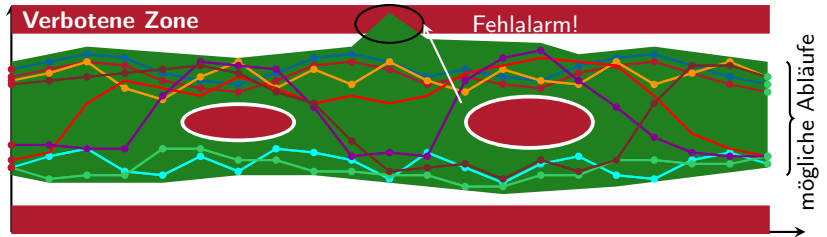






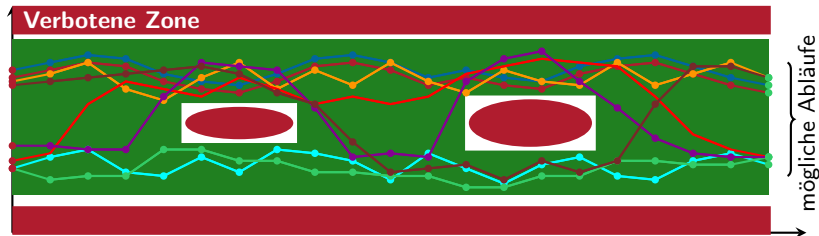
Vollständigkeit und Korrektheit

- Keine potentieller Defekt darf übersehen werden
- ↪ nur so kann die Abwesenheit von Defekten gezeigt werden
 - Ansonsten wäre gegenüber reinem Testen nichts gewonnen



Präzision

- Weitgehende Vermeidung von **Fehlalarmen** (engl. *false alarms*)
 - Synonyme englische Bezeichnung: **false positives**
- Ermöglicht erst eine vollkommen automatisierte Anwendung



Geringe Komplexität

- Berechnung der abstrakten Semantik in akzeptabler Laufzeit
 - Vermeidung der **kombinatorischen Explosion** des Zustandsraums

Reduktion des Zustandsraums ist unumgänglich!

 Fasse verschiedene Zustände geeignet zusammen

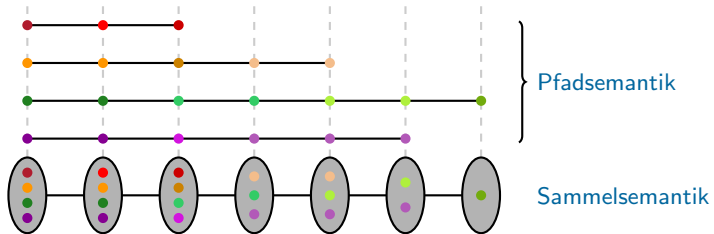
~> **Sammelsemantiken** (s. Folie 15 ff.)

 Betrachte nur den Anfang der Zustandshistorie

~> **Präfixsemantiken** (s. Folie 20 ff.)



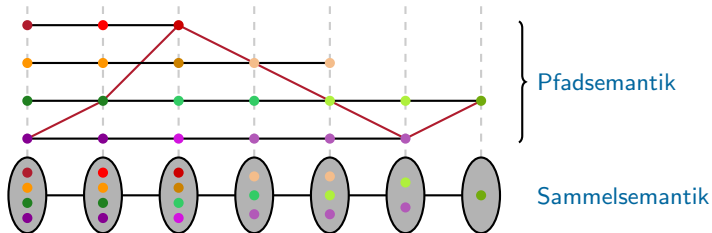
Sammelsemantik (engl. *collecting semantics*)



- Sammelt die Zustände aller Pfade an einem bestimmten Punkt
 - d. h. an einer bestimmten Programmanweisung
 - Aufgrund der Größe, wird sie i. d. R. approximiert



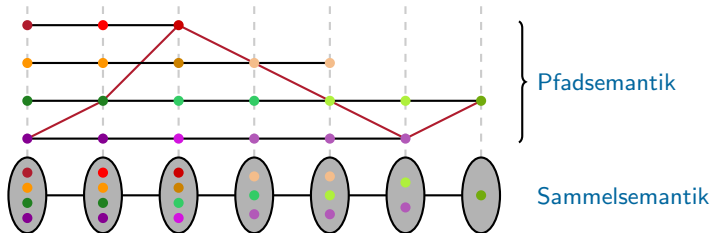
Sammelsemantik (engl. *collecting semantics*)



- Sammelt die Zustände aller Pfade an einem bestimmten Punkt
 - d. h. an einer bestimmten Programmanweisung
 - Aufgrund der Größe, wird sie i. d. R. approximiert
- Das ist eine **verlustbehaftete Abstraktion**
 - Beispiel: Existiert der rote Pfad?
 - Konkrete Semantik \mapsto **Nein**, Sammelsemantik \mapsto ???



Sammelsemantik (engl. *collecting semantics*)



- Sammelt die Zustände aller Pfade an einem bestimmten Punkt
 - d. h. an einer bestimmten Programmanweisung
 - Aufgrund der Größe, wird sie i. d. R. approximiert
- Das ist eine **verlustbehaftete Abstraktion**
 - Beispiel: Existiert der rote Pfad?
 - Konkrete Semantik \mapsto **Nein**, Sammelsemantik \mapsto ???

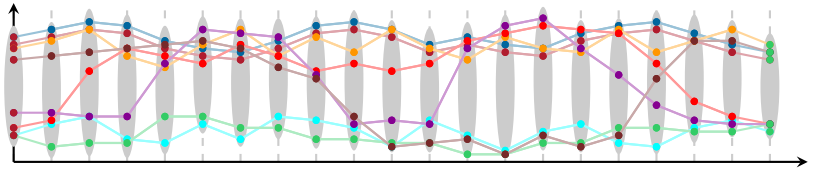


Der **Laufzeitgewinn** wird durch **Unschärfe** erkauft!

- Das Ergebnis „**Weiß nicht ...**“ ist typisch für solche Methoden
- Und die Ursache vieler Vorbehalte ...



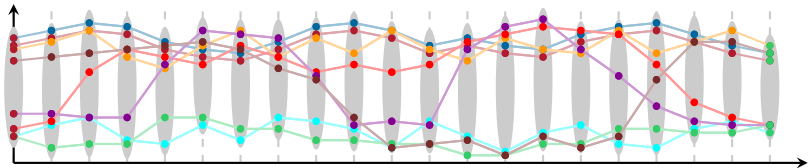
Als Approximation der Sammelsemantik [6, Woche 5]



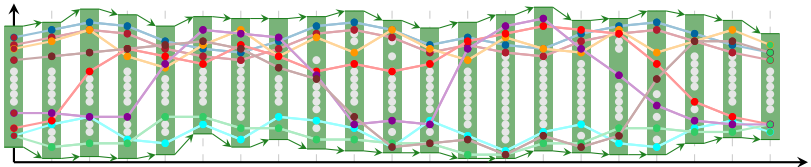
- Die Sammelsemantik verwaltet Zustandsmengen



Als Approximation der Sammelsemantik [6, Woche 5]



- Die Sammelsemantik verwaltet Zustandsmengen
 - ☞ die Intervallabstraktion nur ihre oberen und unteren Schranken
 - Die zu verwaltenden Daten werden dadurch beträchtliche reduziert
 - Allerdings wird auch die Präzision reduziert
- ↪ bestimmte Zustände im approximierten Zustandsraum werden nicht erreicht



Beispiel: Intervallabstraktion für ein C-Programm

```
1 unsigned short x = 1;  
2  
3 while(x < 10000) {  
4     x = x + 1;  
5 }  
6  
7 return x;
```

Die Intervallabstraktion liefert:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = (x_1 \cup x_4) \cap [-\infty, 9999]$

Zeile 4 $x_4 = x_3 \oplus [1, 1]$

Zeile 7 $x_7 = (x_1 \cup x_5) \cap [10000, \infty]$

- Die Intervallabstraktion ist eine **manuell vorgegebene, abstrakte Interpretation** der Semantik der Programmiersprache C
 - C-Programme werden dann **automatisiert darauf abgebildet**
 - z. B. durch einen Übersetzer oder ein statisches Analysewerkzeug
 - Nur Elemente, die den Wertebereich von x betreffen, sind relevant



Beispiel: Intervallabstraktion für ein C-Programm

```
1 unsigned short x = 1;
2
3 while(x < 10000) {
4     x = x + 1;
5 }
6
7 return x;
```

Die Intervallabstraktion liefert:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = (x_1 \cup x_4) \cap [-\infty, 9999]$

Zeile 4 $x_4 = x_3 \oplus [1, 1]$

Zeile 7 $x_7 = (x_1 \cup x_5) \cap [10000, \infty]$

- Die Intervallabstraktion ist eine **manuell vorgegebene, abstrakte Interpretation** der Semantik der Programmiersprache C
 - C-Programme werden dann **automatisiert darauf abgebildet**
 - z. B. durch einen Übersetzer oder ein statisches Analysewerkzeug
 - Nur Elemente, die den Wertebereich von x betreffen, sind relevant
- Dies ist bereits eine **starke Vereinfachung**
 - Angenommen x wäre eingangs nicht bekannt
 - ~ es gäbe 10000 verschiedene Pfade durch den Zustandsraum
 - Nehme eine Schleifenobergrenze **unsigned short** y statt 10000 an
 - ~ es gäbe $\leq (2^{16})^2$ verschiedene Pfade durch den Zustandsraum



```
1 unsigned short x = 1;  
2  
3 while(x < 10000) {  
4     x = x + 1;  
5 }  
6  
7 return x;
```

Die Intervallabstraktion liefert:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = (x_1 \cup x_4) \cap [-\infty, 9999]$

Zeile 4 $x_4 = x_3 \oplus [1, 1]$

Zeile 7 $x_7 = (x_1 \cup x_4) \cap [10000, \infty]$



```
1 unsigned short x = 1;  
2  
3 while(x < 10000) {  
4     x = x + 1;  
5 }  
6  
7 return x;
```

Die Intervallabstraktion liefert:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = (x_1 \cup x_4) \cap [-\infty, 9999]$

Zeile 4 $x_4 = x_3 \oplus [1, 1]$

Zeile 7 $x_7 = (x_1 \cup x_4) \cap [10000, \infty]$

- Approximation durch chaotische Iteration (engl. *chaotic iteration*)

Iteration 1:

Zeile 1 $x_1 = \emptyset$

Zeile 3 $x_3 = \emptyset$

Zeile 4 $x_4 = \emptyset$

Zeile 7 $x_7 = \emptyset$



```
1 unsigned short x = 1;  
2  
3 while(x < 10000) {  
4     x = x + 1;  
5 }  
6  
7 return x;
```

Die Intervallabstraktion liefert:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = (x_1 \cup x_4) \cap [-\infty, 9999]$

Zeile 4 $x_4 = x_3 \oplus [1, 1]$

Zeile 7 $x_7 = (x_1 \cup x_4) \cap [10000, \infty]$

- Approximation durch chaotische Iteration (engl. *chaotic iteration*)

Iteration 1:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = \emptyset$

Zeile 4 $x_4 = \emptyset$

Zeile 7 $x_7 = \emptyset$



```
1 unsigned short x = 1;  
2  
3 while(x < 10000) {  
4     x = x + 1;  
5 }  
6  
7 return x;
```

Die Intervallabstraktion liefert:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = (x_1 \cup x_4) \cap [-\infty, 9999]$

Zeile 4 $x_4 = x_3 \oplus [1, 1]$

Zeile 7 $x_7 = (x_1 \cup x_4) \cap [10000, \infty]$

- Approximation durch chaotische Iteration (engl. *chaotic iteration*)

Iteration 1:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = [1, 1]$

Zeile 4 $x_4 = \emptyset$

Zeile 7 $x_7 = \emptyset$



```
1 unsigned short x = 1;  
2  
3 while(x < 10000) {  
4     x = x + 1;  
5 }  
6  
7 return x;
```

Die Intervallabstraktion liefert:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = (x_1 \cup x_4) \cap [-\infty, 9999]$

Zeile 4 $x_4 = x_3 \oplus [1, 1]$

Zeile 7 $x_7 = (x_1 \cup x_4) \cap [10000, \infty]$

- Approximation durch chaotische Iteration (engl. *chaotic iteration*)

Iteration 1:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = [1, 1]$

Zeile 4 $x_4 = [2, 2]$

Zeile 7 $x_7 = \emptyset$




```
1 unsigned short x = 1;  
2  
3 while(x < 10000) {  
4     x = x + 1;  
5 }  
6  
7 return x;
```

Die Intervallabstraktion liefert:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = (x_1 \cup x_4) \cap [-\infty, 9999]$

Zeile 4 $x_4 = x_3 \oplus [1, 1]$

Zeile 7 $x_7 = (x_1 \cup x_4) \cap [10000, \infty]$

■ Approximation durch chaotische Iteration (engl. *chaotic iteration*)

Iteration 1:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = [1, 1]$

Zeile 4 $x_4 = [2, 2]$

Zeile 7 $x_7 = \emptyset$

Iteration 2:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = [1, 2]$

Zeile 4 $x_4 = [2, 2]$

Zeile 7 $x_7 = \emptyset$



```
1 unsigned short x = 1;  
2  
3 while(x < 10000) {  
4     x = x + 1;  
5 }  
6  
7 return x;
```

Die Intervallabstraktion liefert:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = (x_1 \cup x_4) \cap [-\infty, 9999]$

Zeile 4 $x_4 = x_3 \oplus [1, 1]$

Zeile 7 $x_7 = (x_1 \cup x_4) \cap [10000, \infty]$

■ Approximation durch chaotische Iteration (engl. *chaotic iteration*)

Iteration 1:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = [1, 1]$

Zeile 4 $x_4 = [2, 2]$

Zeile 7 $x_7 = \emptyset$

Iteration 2:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = [1, 2]$

Zeile 4 $x_4 = [2, 3]$

Zeile 7 $x_7 = \emptyset$



```
1 unsigned short x = 1;  
2  
3 while(x < 10000) {  
4     x = x + 1;  
5 }  
6  
7 return x;
```

Die Intervallabstraktion liefert:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = (x_1 \cup x_4) \cap [-\infty, 9999]$

Zeile 4 $x_4 = x_3 \oplus [1, 1]$

Zeile 7 $x_7 = (x_1 \cup x_4) \cap [10000, \infty]$

- Approximation durch chaotische Iteration (engl. *chaotic iteration*)

Iteration 3:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = [1, 3]$

Zeile 4 $x_4 = [2, 3]$

Zeile 7 $x_7 = \emptyset$



```
1 unsigned short x = 1;  
2  
3 while(x < 10000) {  
4     x = x + 1;  
5 }  
6  
7 return x;
```

Die Intervallabstraktion liefert:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = (x_1 \cup x_4) \cap [-\infty, 9999]$

Zeile 4 $x_4 = x_3 \oplus [1, 1]$

Zeile 7 $x_7 = (x_1 \cup x_4) \cap [10000, \infty]$

- Approximation durch chaotische Iteration (engl. *chaotic iteration*)

Iteration 3:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = [1, 3]$

Zeile 4 $x_4 = [2, 4]$

Zeile 7 $x_7 = \emptyset$



```
1 unsigned short x = 1;  
2  
3 while(x < 10000) {  
4     x = x + 1;  
5 }  
6  
7 return x;
```

Die Intervallabstraktion liefert:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = (x_1 \cup x_4) \cap [-\infty, 9999]$

Zeile 4 $x_4 = x_3 \oplus [1, 1]$

Zeile 7 $x_7 = (x_1 \cup x_4) \cap [10000, \infty]$

■ Approximation durch chaotische Iteration (engl. *chaotic iteration*)

Iteration 3:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = [1, 3]$

Zeile 4 $x_4 = [2, 4]$

Zeile 7 $x_7 = \emptyset$

viele, viele Iterationen später:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = [1, 9999]$

Zeile 4 $x_4 = [2, 9999]$

Zeile 7 $x_7 = \emptyset$



```
1 unsigned short x = 1;  
2  
3 while(x < 10000) {  
4     x = x + 1;  
5 }  
6  
7 return x;
```

Die Intervallabstraktion liefert:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = (x_1 \cup x_4) \cap [-\infty, 9999]$

Zeile 4 $x_4 = x_3 \oplus [1, 1]$

Zeile 7 $x_7 = (x_1 \cup x_4) \cap [10000, \infty]$

- Approximation durch chaotische Iteration (engl. *chaotic iteration*)

Iteration 3:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = [1, 3]$

Zeile 4 $x_4 = [2, 4]$

Zeile 7 $x_7 = \emptyset$

viele, viele Iterationen später:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = [1, 9999]$

Zeile 4 $x_4 = [2, 10000]$

Zeile 7 $x_7 = \emptyset$



```
1 unsigned short x = 1;  
2  
3 while (x < 10000) {  
4     x = x + 1;  
5 }  
6  
7 return x;
```

Die Intervallabstraktion liefert:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = (x_1 \cup x_4) \cap [-\infty, 9999]$

Zeile 4 $x_4 = x_3 \oplus [1, 1]$

Zeile 7 $x_7 = (x_1 \cup x_4) \cap [10000, \infty]$

- Approximation durch chaotische Iteration (engl. *chaotic iteration*)

Iteration 3:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = [1, 3]$

Zeile 4 $x_4 = [2, 4]$

Zeile 7 $x_7 = \emptyset$

viele, viele Iterationen später:

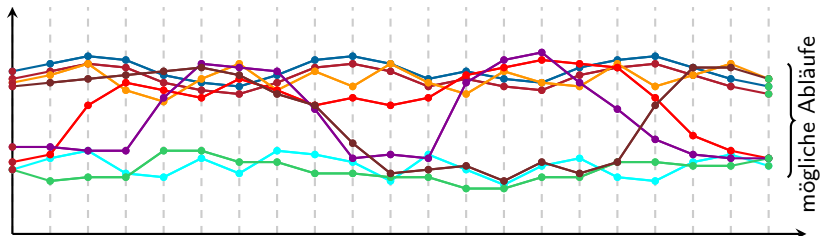
Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = [1, 9999]$

Zeile 4 $x_4 = [2, 10000]$

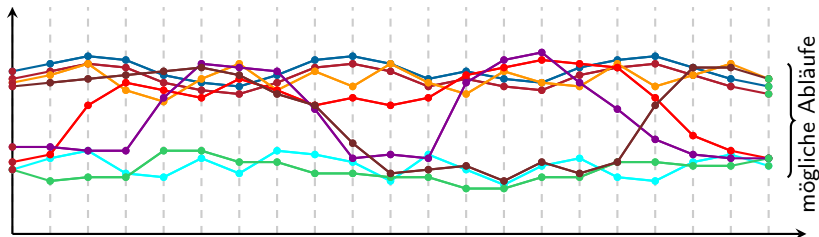
Zeile 7 $x_7 = [10000, 10000]$





- Betrachte durch ein **Transitionssystem** beschriebene **Programmpfade**
 - Ausgehend von ausgezeichneten Startzuständen,
 - Beschreiben sie eine (unendliche) Abfolge von **Programmzuständen**,
 - Deren Reihenfolge durch die Übergangsrelation bestimmt wird.
- ~ die Gesamtheit dieser Programmpfade heißt **Pfadsemantik**
- Wie die konkrete Programmsemantik ist sie **nicht berechenbar**.



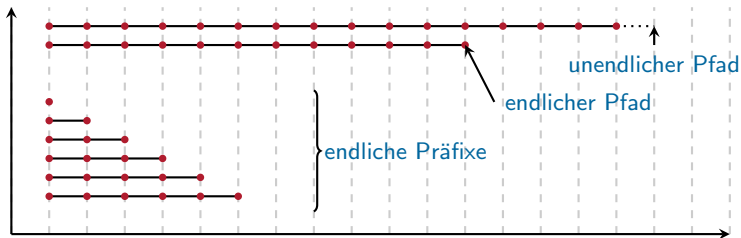


- Betrachte durch ein **Transitionssystem** beschriebene **Programmpfade**
 - Ausgehend von ausgezeichneten Startzuständen,
 - Beschreiben sie eine (unendliche) Abfolge von **Programmzuständen**,
 - Deren Reihenfolge durch die Übergangsrelation bestimmt wird.

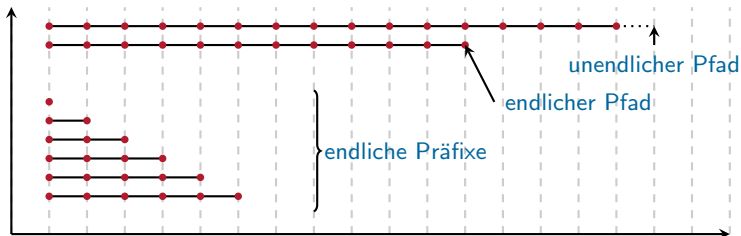
↪ die Gesamtheit dieser Programmpfade heißt **Pfadsemantik**

 - Wie die konkrete Programmsemantik ist sie **nicht berechenbar**.
- Reduktion der Komplexität durch **Abstraktion**
 - Unendliche Pfade \leadsto (endliche) **Pfadpräfixe**





- Pfadsemantiken enthalten alle endlichen und unendlichen Pfade
- Pfadpräfixe enthalten nur die Anfänge dieser Pfade



- Pfadsemantiken enthalten alle endlichen und unendlichen Pfade
 - Pfadpräfixe enthalten nur die Anfänge dieser Pfade

☞ das ist eine **verlustbehaftete Abstraktion**

- Beispiel: betrachte Worte der Sprache $a^n b$
 - Frage: Gibt es Worte mit unendlich vielen aufeinanderfolgenden 'a'?
 - Pfadsemantik: $\{a^n b | n \geq 0\} \mapsto \text{Nein}$
 - Pfadpräfixe: $\{a^n | n \geq 0\} \cup \{a^n b | n \geq 0\} \mapsto ???$



- Menge der Prafixe ist rekursiv:

$$\begin{aligned} \text{Prafixe} = & \{x \mid x \text{ ist Startzustand}\} \cup \\ & \{x_1 \rightarrow^* x_2 \rightarrow x_3 \mid x_1 \rightarrow^* x_2 \in \text{Prafixe} \wedge x_2 \rightarrow x_3 \in \rightarrow\} \end{aligned}$$

- Zu losen ist die Fixpunktiteration $\text{Prafixe} = F(\text{Prafixe})$

- ublicherweise besitzt diese Gleichung mehrere Losungen
- ~ ordne die Losungen nach der **Teilmengenbeziehung** \subseteq
- ~ wahle die kleinste Teilmenge als Losung
- ~ **least fixpoint prefix trace semantics**



- Menge der Prafixe ist rekursiv:

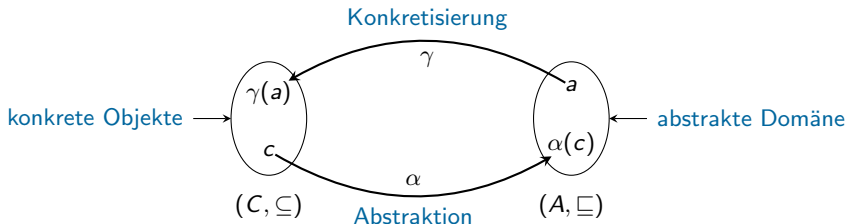
$$\begin{aligned} \text{Prafixe} = \{x \mid x \text{ ist Startzustand}\} \cup \\ \{x_1 \rightarrow^* x_2 \rightarrow x_3 \mid x_1 \rightarrow^* x_2 \in \text{Prafixe} \wedge x_2 \rightarrow x_3 \in \rightarrow\} \end{aligned}$$

- Zu losen ist die Fixpunktiteration $\text{Prafixe} = F(\text{Prafixe})$
 - ublicherweise besitzt diese Gleichung mehrere Losungen
 - ~> ordne die Losungen nach der **Teilmengenbeziehung** \subseteq
 - ~> wahle die kleinste Teilmenge als Losung
 - ~> **least fixpoint prefix trace semantics**
- Vereinfachungen ermoglichen **effektive, iterative Analysealgorithmen**
 - Vereinfachung im Sinne von Abstraktion bzw. Approximation
 - ~> man muss nur noch die Prafixe betrachten
 - Nicht mehr die vollstandigen (evtl. unendlichen) Pfade



- 1 Überblick
- 2 Problemstellung
 - Beispiel
 - Konkrete Programmsemantik
- 3 Abstrakte Interpretation
 - Abstrakte Semantik
 - Sammelsemantiken
 - Präfixsemantiken
- 4 Mathematische Grundlagen (Selbststudium)
- 5 Zusammenfassung





- Wähle eine **abstrakte Domäne** (engl. *abstract domain*)
 - Ersetzt die Menge konkreter Objekte S durch ihre Abstraktion $\alpha(S)$
 - Verschiedene Domänen unterscheiden sich hinsichtlich ihrer Präzision
 - Vorzeichen, **Intervalle**, Oktagon, Polyhedra, ...
- **Abstraktionsfunktion** α (engl. *abstraction function*)
 - Bildet die Menge konkrete Objekte auf ihre abstrakte Interpretation ab
- **Konkretisierungsfunktion** γ (engl. *concretization function*)
 - Bildet die Menge abstrakter Objekte auf konkrete Objekte ab





Approximation von f durch die abstrakte Funktion f'





Approximation von f durch die abstrakte Funktion f'

■ Häufig verwendet man **Galoiseinbettungen**

- Diese sind Galoisverbindungen $(C, \subseteq) \xleftrightarrow[\alpha]{\gamma} (A, \sqsubseteq)$
- Mit der Eigenschaft $\alpha(\gamma(a)) = a$
 - Konkretisierung gefolgt von Abstraktion impliziert keinen Präzisionsverlust





Approximation von f durch die abstrakte Funktion f'

- Häufig verwendet man **Galoiseinbettungen**

- Diese sind Galoisverbindungen $(C, \subseteq) \xrightleftharpoons[\alpha]{\gamma} (A, \sqsubseteq)$
- Mit der Eigenschaft $\alpha(\gamma(a)) = a$
 - Konkretisierung gefolgt von Abstraktion impliziert keinen Präzisionsverlust

- **Abstrakte Interpretation** nutzt diese Eigenschaften

- Statt die konkrete Funktion $f(c)$ zu berechnen
- Kann man sie annähern, indem
 - Man die abstrakte Funktion f' auf die Abstraktion $\alpha(c)$ anwendet
 - Und das Ergebnis $f'(\alpha(c))$ wieder konkretisiert



Warum funktioniert das eigentlich ... ?

- Wann ist eine Abstraktion **korrekt**?

↪ Wenn sie durch eine **Galoisverbindung** beschrieben wird!

↪ **OK!**



Warum funktioniert das eigentlich ... ?

- Wann ist eine Abstraktion **korrekt**?

↪ Wenn sie durch eine **Galoisverbindung** beschrieben wird!

↪ **OK!**

- Fixpunkte ... wer sagt, dass die Iteration überhaupt **konvergiert**?

↪ **Aufsteigende Kettenbedingung!**



Warum funktioniert das eigentlich ... ?

- Wann ist eine Abstraktion **korrekt**?

↪ Wenn sie durch eine **Galoisverbindung** beschrieben wird!

↪ **OK!**

- Fixpunkte ... wer sagt, dass die Iteration überhaupt **konvergiert**?

↪ **Aufsteigende Kettenbedingung!**

↪ **???**



Warum funktioniert das eigentlich ... ?

- Wann ist eine Abstraktion **korrekt**?

↪ Wenn sie durch eine **Galoisverbindung** beschrieben wird!

↪ **OK!**

- Fixpunkte ... wer sagt, dass die Iteration überhaupt **konvergiert**?

↪ **Aufsteigende Kettenbedingung!**

↪ **???**

- Das waren ziemlich viele Iterationen ... geht das auch **schneller**?

↪ **Widening-/Narrowing-Operatoren** helfen!



Warum funktioniert das eigentlich ... ?

- Wann ist eine Abstraktion **korrekt**?

~> Wenn sie durch eine **Galoisverbindung** beschrieben wird! ~> **OK!**

- Fixpunkte ... wer sagt, dass die Iteration überhaupt **konvergiert**?

~> **Aufsteigende Kettenbedingung!** ~> **???**

- Das waren ziemlich viele Iterationen ... geht das auch **schneller**?

~> **Widening-/Narrowing-Operatoren** helfen! ~> **???**



Warum funktioniert das eigentlich ... ?

- Wann ist eine Abstraktion **korrekt**?

↪ Wenn sie durch eine **Galoisverbindung** beschrieben wird! ↪ **OK!**

- Fixpunkte ... wer sagt, dass die Iteration überhaupt **konvergiert**?

↪ **Aufsteigende Kettenbedingung!** ↪ **???**

- Das waren ziemlich viele Iterationen ... geht das auch **schneller**?

↪ **Widening-/Narrowing-Operatoren** helfen! ↪ **???**

- **Jetzt:** Grundlegende mathematische Zusammenhänge erfassen!

- Was ist das und was hat es mit abstrakter Interpretation zu tun?

- **Nicht:** Warum ist das korrekt?

- Keine Beweisführung ...



- Konkrete und abstrakte Domänen sind **partiell geordnete Mengen**!



- Konkrete und abstrakte Domänen sind **partiell geordnete Mengen**!

Partiell geordnete Mengen (engl. *partially ordered sets*)

Eine **partiell geordnete Menge** ist ein Tupel (S, \sqsubseteq) :

- S ist eine Menge,
- $\sqsubseteq \subseteq S \times S$ ist eine **Ordnungsrelation** mit folgenden Eigenschaften:

Reflexiv $\forall x \in S : x \sqsubseteq x$

Antisymmetrisch $\forall x, y \in S : x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$

Transitiv $\forall x, y, z \in S : x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$



- Konkrete und abstrakte Domänen sind **partiell geordnete Mengen**!

Partiell geordnete Mengen (engl. *partially ordered sets*)

Eine **partiell geordnete Menge** ist ein Tupel (S, \sqsubseteq) :

- S ist eine Menge,
- $\sqsubseteq \subseteq S \times S$ ist eine **Ordnungsrelation** mit folgenden Eigenschaften:

Reflexiv $\forall x \in S : x \sqsubseteq x$

Antisymmetrisch $\forall x, y \in S : x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$

Transitiv $\forall x, y, z \in S : x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$

- Beispiele:
 - (\mathbb{N}, \leq) ist ein partiell geordnete Menge
 - $(\mathcal{P}(S), \subseteq)$ ist ein partiell geordnete Menge



Obere und untere Schranken

- Sei (S, \sqsubseteq) eine partiell geordnete Menge

Obere Schranke (engl. *upper bound*)

$x \in S$ eine obere Schranke von $P \subseteq S \Leftrightarrow y \in P : y \sqsubseteq x$

☞ analog: untere Schranke (engl. *lower bound*)



Obere und untere Schranken

- Sei (S, \sqsubseteq) eine partiell geordnete Menge

Obere Schranke (engl. *upper bound*)

$x \in S$ eine obere Schranke von $P \subseteq S \Leftrightarrow y \in P : y \sqsubseteq x$

☞ analog: untere Schranke (engl. *lower bound*)

Kleinste obere Schranke (engl. *least upper bound*)

$x \in S$ ist eine kleinste obere Schranke von $P \subseteq S \Leftrightarrow$

- x ist eine obere Schranke von P und
- x ist kleiner als alle oberen Schranken von P :

$$\forall y \in S : (\forall z \in P : z \sqsubseteq y) \Rightarrow x \sqsubseteq y$$

☞ analog: größte untere Schranke (engl. *greatest lower bound*)



Vollständiger Verband (engl. *complete lattice*)

Ein **vollständiger Verband** ist eine partiell geordnete Menge $(S, \sqsubseteq, \perp, \top, \sqcup, \sqcap)$ mit folgenden Eigenschaften:

- (S, \sqsubseteq) ist eine partiell geordnete Menge
- Für jede Teilmenge $P \subseteq S$ existiert eine
 - Eine kleinste obere Schranke $\sqcup P$ und
 - Eine größte untere Schranke $\sqcap P$
- $\perp = \sqcap S$ heißt **Infimum** von S
- $\top = \sqcup S$ heißt **Supremum** von S

■ Beispiele:

- $(\mathcal{P}(S), \subseteq, \emptyset, S, \cup, \cap)$ ist ein vollständiger Verband
- $(\mathbb{Z} \cup \{-\infty, +\infty\}, \leq, -\infty, +\infty, \max, \min)$ ist ein vollständiger Verband
 - Die Menge der ganzen Zahlen erweitert um $-\infty$ und $+\infty$



Terminierung der Fixpunktiteration

- Möglichkeit 1: aufsteigende Kettenbedingung ist erfüllt
 - ↪ aufsteigende Ketten sind endlich
 - ↪ Fixpunktiteration terminiert



Terminierung der Fixpunktiteration

- **Möglichkeit 1:** aufsteigende Kettenbedingung **ist erfüllt**
 - ↪ aufsteigende Ketten sind endlich
 - ↪ Fixpunktiteration terminiert
- **Möglichkeit 2:** aufsteigende Kettenbedingung **ist nicht erfüllt**
 - ↪ Terminierung kann durch einen **Widening-Operator** erzwungen werden



Terminierung der Fixpunktiteration

- **Möglichkeit 1:** aufsteigende Kettenbedingung **ist erfüllt**
 - ↪ aufsteigende Ketten sind endlich
 - ↪ Fixpunktiteration terminiert
- **Möglichkeit 2:** aufsteigende Kettenbedingung **ist nicht erfüllt**
 - ↪ Terminierung kann durch einen **Widening-Operator** erzwungen werden

Widening-Operator

Sei V ein Verband, ein **Widening-Operator** $\nabla : V \times V \mapsto V$ ist eine Abbildung für die gilt:

$$\forall x, y \in V : x \sqsubseteq x \nabla y \wedge y \sqsubseteq x \nabla y$$

- Sicher Abschätzung der Elemente x und y nach oben durch $x \nabla y$
- Ermöglicht auch eine Beschleunigung der Fixpunktiteration
 - Widening-Operator $\nabla \approx$ Bestimmung der kleinsten oberen Schranke
 - In vollständigen Verbänden mit aufsteigender Kettenbedingung



Beispiel: Intervallabstraktion – nun mit Widening

```
1 unsigned short x = 1;
2
3 while(x < 10000) {
4     x = x + 1;
5 }
6
7 return x;
```

Die Intervallabstraktion liefert:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = (x_1 \nabla x_4) \cap [-\infty, 9999]$

Zeile 4 $x_4 = x_3 \oplus [1, 1]$

Zeile 7 $x_7 = (x_1 \nabla x_4) \cap [10000, \infty]$



Beispiel: Intervallabstraktion – nun mit Widening

```
1 unsigned short x = 1;  
2  
3 while (x < 10000) {  
4     x = x + 1;  
5 }  
6  
7 return x;
```

Die Intervallabstraktion liefert:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = (x_1 \nabla x_4) \cap [-\infty, 9999]$

Zeile 4 $x_4 = x_3 \oplus [1, 1]$

Zeile 7 $x_7 = (x_1 \nabla x_4) \cap [10000, \infty]$

■ Approximation mit Hilfe des Widening-Operators

Iteration 1:

Zeile 1 $x_1 = \emptyset$

Zeile 3 $x_3 = \emptyset$

Zeile 4 $x_4 = \emptyset$

Zeile 7 $x_7 = \emptyset$



Beispiel: Intervallabstraktion – nun mit Widening

```
1 unsigned short x = 1;  
2  
3 while (x < 10000) {  
4     x = x + 1;  
5 }  
6  
7 return x;
```

Die Intervallabstraktion liefert:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = (x_1 \nabla x_4) \cap [-\infty, 9999]$

Zeile 4 $x_4 = x_3 \oplus [1, 1]$

Zeile 7 $x_7 = (x_1 \nabla x_4) \cap [10000, \infty]$

■ Approximation mit Hilfe des Widening-Operators

Iteration 1:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = \emptyset$

Zeile 4 $x_4 = \emptyset$

Zeile 7 $x_7 = \emptyset$



Beispiel: Intervallabstraktion – nun mit Widening

```
1 unsigned short x = 1;  
2  
3 while (x < 10000) {  
4     x = x + 1;  
5 }  
6  
7 return x;
```

Die Intervallabstraktion liefert:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = (x_1 \nabla x_4) \cap [-\infty, 9999]$

Zeile 4 $x_4 = x_3 \oplus [1, 1]$

Zeile 7 $x_7 = (x_1 \nabla x_4) \cap [10000, \infty]$

■ Approximation mit Hilfe des Widening-Operators

Iteration 1:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = [1, 1]$

Zeile 4 $x_4 = \emptyset$

Zeile 7 $x_7 = \emptyset$



Beispiel: Intervallabstraktion – nun mit Widening

```
1 unsigned short x = 1;
2
3 while(x < 10000) {
4     x = x + 1;
5 }
6
7 return x;
```

Die Intervallabstraktion liefert:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = (x_1 \nabla x_4) \cap [-\infty, 9999]$

Zeile 4 $x_4 = x_3 \oplus [1, 1]$

Zeile 7 $x_7 = (x_1 \nabla x_4) \cap [10000, \infty]$

■ Approximation mit Hilfe des Widening-Operators

Iteration 1:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = [1, 1]$

Zeile 4 $x_4 = [2, 2]$

Zeile 7 $x_7 = \emptyset$



```
1 unsigned short x = 1;  
2  
3 while (x < 10000) {  
4     x = x + 1;  
5 }  
6  
7 return x;
```

Die Intervallabstraktion liefert:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = (x_1 \nabla x_4) \cap [-\infty, 9999]$

Zeile 4 $x_4 = x_3 \oplus [1, 1]$

Zeile 7 $x_7 = (x_1 \nabla x_4) \cap [10000, \infty]$

■ Approximation mit Hilfe des Widening-Operators

Iteration 1:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = [1, 1]$

Zeile 4 $x_4 = [2, 2]$

Zeile 7 $x_7 = \emptyset$

Iteration 2:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = [1, 9999]$

Zeile 4 $x_4 = [2, 2]$

Zeile 7 $x_7 = \emptyset$



```
1 unsigned short x = 1;  
2  
3 while (x < 10000) {  
4     x = x + 1;  
5 }  
6  
7 return x;
```

Die Intervallabstraktion liefert:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = (x_1 \nabla x_4) \cap [-\infty, 9999]$

Zeile 4 $x_4 = x_3 \oplus [1, 1]$

Zeile 7 $x_7 = (x_1 \nabla x_4) \cap [10000, \infty]$

■ Approximation mit Hilfe des Widening-Operators

Iteration 1:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = [1, 1]$

Zeile 4 $x_4 = [2, 2]$

Zeile 7 $x_7 = \emptyset$

Iteration 2:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = [1, 9999]$

Zeile 4 $x_4 = [2, 10000]$

Zeile 7 $x_7 = \emptyset$



Beispiel: Intervallabstraktion – nun mit Widening

```
1 unsigned short x = 1;
2
3 while (x < 10000) {
4     x = x + 1;
5 }
6
7 return x;
```

Die Intervallabstraktion liefert:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = (x_1 \nabla x_4) \cap [-\infty, 9999]$

Zeile 4 $x_4 = x_3 \oplus [1, 1]$

Zeile 7 $x_7 = (x_1 \nabla x_4) \cap [10000, \infty]$

■ Approximation mit Hilfe des Widening-Operators

Iteration 1:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = [1, 1]$

Zeile 4 $x_4 = [2, 2]$

Zeile 7 $x_7 = \emptyset$

Iteration 2:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = [1, 9999]$

Zeile 4 $x_4 = [2, 10000]$

Zeile 7 $x_7 = [10000, 10000]$



Beispiel: Intervallabstraktion – nun mit Widening

```
1 unsigned short x = 1;
2
3 while (x < 10000) {
4     x = x + 1;
5 }
6
7 return x;
```

Die Intervallabstraktion liefert:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = (x_1 \nabla x_4) \cap [-\infty, 9999]$

Zeile 4 $x_4 = x_3 \oplus [1, 1]$

Zeile 7 $x_7 = (x_1 \nabla x_4) \cap [10000, \infty]$

■ Approximation mit Hilfe des Widening-Operators

Iteration 1:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = [1, 1]$

Zeile 4 $x_4 = [2, 2]$

Zeile 7 $x_7 = \emptyset$

Iteration 2:

Zeile 1 $x_1 = [1, 1]$

Zeile 3 $x_3 = [1, 9999]$

Zeile 4 $x_4 = [2, 10000]$

Zeile 7 $x_7 = [10000, 10000]$

■ ➡ Konvergenz in der 2. Iteration



- 1 Überblick
- 2 Problemstellung
 - Beispiel
 - Konkrete Programmsemantik
- 3 Abstrakte Interpretation
 - Abstrakte Semantik
 - Sammelsemantiken
 - Präfixsemantiken
- 4 Mathematische Grundlagen (Selbststudium)
- 5 Zusammenfassung



Konkrete Programmsemantik ist **nicht berechenbar**

- Approximation durch eine **abstrakte Semantik**
 - Korrektheit der Approximation ist entscheidend
 - Nur so kann man einen **Sicherheitsnachweis** führen
 - Die Approximation muss präzise sein
 - Nur so kann man **Fehlalarme** vermeiden
 - Die Approximation darf nicht zu komplex sein
 - Nur so kann sie **effizient berechnet** werden

Transitionssystem beschreiben Programme

- **Pfadsemantiken** beschreiben die konkrete Programmsemantik
- Approximation durch **Pfadpräfixe** und **Sammelsemantik**
 - ↪ abstrakte Interpretation approximiert die Sammelsemantik

Mathematische Grundlagen abstrakter Interpretation

- (vollständig) partiell geordnete Mengen, Verbände
- Galoiseinbettungen, lokale konsistente Funktionen, Widening
- Intervallabstraktion



- [1] COUSOT, P. :
Semantic foundations of program analysis.
In: *Program flow analysis: theory and applications* 10 (1981), S. 303–342

- [2] COUSOT, P. :
Abstract Interpretation.
<http://web.mit.edu/16.399/www/>, 2005

- [3] COUSOT, P. ; COUSOT, R. :
Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints.
In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*.
ACM (POPL '77), 238–252

- [4] COUSOT, P. ; COUSOT, R. :
Abstract interpretation frameworks.
In: *Journal of Logic and Computation* 2 (1992), Nr. 4, S. 511–547



- [5] COUSOT, P. ; COUSOT, R. :
Abstract Interpretation and Application to Logic Programs.
In: *Journal of Logic Programming* 13 (1992), Jul., Nr. 2-3, S. 103–179.
[http://dx.doi.org/10.1016/0743-1066\(92\)90030-7](http://dx.doi.org/10.1016/0743-1066(92)90030-7). –
DOI 10.1016/0743-1066(92)90030-7. –
ISSN 0743-1066
- [6] MIDTGAARD, J. :
Abstract Interpretation.
<http://www.cs.au.dk/~jmi/AbsInt/>, 2012

