7. Compositional Shape Analysis – Towards Facebook Infer

basierend auf "Compositional Shape Analysis by means of Bi-Abduction"

Christiano Calcagno, Dino DiStefano, Peter O'Hearn, Hongseok Yang

POPL'09

Ziel: Betrachte rekursive Programme.
Berechne Menge an SL-Tripeln für jede Funktion.
Diese Tripel überapproximieren die möglichen Heap-Änderungen.
Shape-Analyse = Analyse der Gestalt des Heaps.

Kompositionalität: Nennen eine Programmanalyse kompositionell,
falls sich das Analyseergebnis eines zusammengesetzten Programms
aus den Analyseergebnissen der Teile berechnen lässt.
Warum?

Kein Kontext: Analysiere Programmteile ohne ihren Kontext.
Kontext ist noch nicht geschrieben oder aber riesig.
In beiden Fällen bräuchte man Mocks -> aufwendig.

Scalability: Shape-Analysen sind teuer.
Ohne Kompositionalität skalieren die nicht.

Parallelisierung: Führe Analysen verschiedener Funktionen
unabhängig voneinander parallel aus.

Inkrementelle Berechnung: Speichere das Analyseergebnis ab.
Wenn sich eine Funktion ändert,
update nur das Analyseergebnis dieser Funktion.

Graceful Imprecision: Für jede abstrakte Domäne gibt es Programme,
auf denen sie unpräzise rechnet.

Mit einer kompositionellen Analyse,
lassen sich die Domainen je nach Funktion austauschen.
Außerdem machen Vorbedingungen die Analyse präziser.

7.1 Überblick

Ansatz: • Wir analysieren eine Funktion in Isolation.
Wenn wir an eine Stelle gelangen, an der wir nicht genug Information haben, um fortzufahren, führen wir einen Abduktionsschritt aus, um die fehlende Information zu ermitteln.
Die fehlende Information wird nun an den Anfang unserer Funktion propagiert und dort der Vorbedingung hinzugefügt.

• Unter welchen Umständen haben wir nicht genug Information?
↳ Wenn wir einen Pointer dereferenzieren wollen, aber nicht wissen, dass er $\neq$ NULL ist.
↳ Wenn wir eine andere Funktion aufrufen wollen, aber nicht deren Vorbedingung vorliegen haben.

• Was macht Abduktion?
↳ Es löst das folgende algorithmische Problem:

ABD:
Gegeben: SL-Assertions A und B.
Aufgabe: Berechne SL-Assertion X so, dass $A * X \vdash B$.

↳ Abduktion berechnet also fehlende Prämissen.

Illustration:

Angenommen bei der Analyse einer Methode erreichen wir

$$\vdots$$

$$\{A\}$$

$$f();$$

Wir haben also eine Zusicherung, sagen wir

$$A = x \mapsto 0,$$

vorliegen und möchten Funktion $f()$ aufrufen. Funktion $f()$ hat die Spezifikation

$$\{P\}\ f()\ \{Q\}$$

mit

$$P = list(x) * list(y).$$

Dann ist P keine Folgerung aus A.

Abduktion erlaubt uns nun folgenden Schluss:

"Wenn wir $list(y)$ zu A hinzufügen (mittels $*$), dann folgt P aus $A * list(y)$."

Beispiel:

Betrachte:

```
void merge(list_nd* x, list_nd* y){
    PRE: list(x) * list(y)
    POST: list(x)
}
```

Wir analysieren nun eine Funktion $p(y)$, die merge aufruft.

```
list_nd* p(list_nd* y){ // Inferred PRE: list(y)
    list_nd* x = new list_nd();
  {x ↦ 0}
    merge(x, y);
  {list(x)}
    return x;
} // Inferred POST: list(ret)
```

Beachte: list(y) ist genau die Menge an Zuständen, von denen aus das Programm ohne Speicherfehler läuft.

Bi-Abduktion:

Abduktion berechnet fehlende Teile des Zustands, den Anti-Frame.
Wir müssen auch berechnen, welche Teile des Zustands von einer Funktion nicht geändert werden, den Frame.
Wir müssen daher ein allgemeineres Problem lösen.

BI-ABD:

Gegeben: SL-Assertions A und B.

Aufgabe: Berechne SL-Assertions X und Y so, dass

$$\boxed{A * \underbrace{X}_{\text{Anti-Frame}} \vdash B * \underbrace{Y}_{\text{Frame}}}.$$

Beispiel:

```
list_nd* q(list_nd* y){ // Inferred PRE: list(y)
    list_nd* x = new list_nd();
  {x ↦ 0}
```

list_nd* z = new list_nd();

$\{x \mapsto 0 * z \mapsto 0\}$

merge(x, y);

$\{list(x) * z \mapsto 0\}$

merge(x, z);

$\{list(x)\}$

return x;

} // Inferred POST: list(ret)

Hier findet ein Bi-Abduktionsschritt statt:

$$x \mapsto 0 * z \mapsto 0 * X \vdash list(x) * list(y) * Y.$$

Eine Lösung ist

$X = list(y)$, der Anti-Frame,

$Y = z \mapsto 0$, der Frame.

Bemerkung:

Das Ziel sind lokale / möglichst kurze Spezifikationen.

Warum?

↳ Shape-Analysen werden riesig.

↳ Kompaktere Spezifikationen sind allgemeiner.

Wie hilft Bi-Abduktion?

↳ Die Berechnung von Frames erlaubt die Verwendung kompakter Spezifikationen (deren Wiederverwendung in einem größeren Stück)

↳ Die Berechnung von Anti-Frames dient der Synthese kompakter Spezifikationen.

7.2 Rekursive Programme

Wir erweitern unsere Programme um Funktionsaufrufe:

$$c ::= \underbrace{\ldots}_{\text{wie bisher}} \mid x := f(\bar{a}),$$

wobei $\bar{a} = a_1, \ldots, a_n$ für eine Folge von Ausdrücken steht.

Wir schreiben auch $\bar{x}$ für $x_1 \ldots x_n$.

Ein <u>rekursives Programm</u> ist eine Folge von Funktionsdefinitionen:

$$p ::= f(\bar{x})\ \{\ \underline{\text{local}}\ \bar{y};\ c;\ \underline{\text{return}}\ a;\ \} \quad \mid \quad p;p$$

Wir nehmen an, Funktionen greifen nur auf Variablen aus $\bar{x}$ und $\bar{y}$ zu.

Globale Variablen gibt es nicht.

Der Einfachheit halber wird nur ein Wert zurückgegeben.

7.3 Shape-Analyse

<u>Ziel</u>: Gegeben ein rekursives Programm p mit Funktionsmenge F wollen wir für jede Funktion $f \in F$ eine <u>Spec-Table</u>

$$T(f) : SH \to P(SH)$$

berechnen.

Dabei ist SH eine eingeschränkte Klasse von SL-Assertions, sogenannte <u>Symbolic-Heaps</u> (im Grunde nur $\wedge$ über Pure-Assertions und $*$ über points-to).

Für jeden Symbolic-Heaps P

mit

$$[T(f)](P) = \{Q_1, \ldots, Q_n\}$$

ist die intendierte Bedeutung

$$\boxed{\models \{P\}\ f(\bar{x})\ \{Q_1 \vee \ldots \vee Q_n\}}\ .$$

<u>Ansatz (zur Berechnung der Spec-Tables):</u>

Wir berechnen eine geeignete Semantik des rekursiven Programms.

Die Idee wird bei der Behandlung einzelner Befehle com klar.

Deren Semantik ist vom Typ

$$[\![com]\!]_T : \mathbb{P}(SH \times SH) \longrightarrow \mathbb{P}(SH \times SH),$$

es wird also eine <u>Relation zwischen Symbolic-Heaps</u> in eine andere Relation zwischen Symbolic-Heaps transformiert.

Gegeben ein Paar $(P, H) \in SH \times SH$ liefert die Semantik $(P * M, H') \in SH \times SH$, in folgender Situation:

- P ist die Vorbedingung, H ist der aktuelle Heap, dazwischen liegen ggf. schon einige Befehle.
- Wenn M zu P hinzugefügt wird, kann der Befehl com H zu H' transformieren.

Beachte, dass die Semantik abhängig ist
von den aktuellen Spezifikationen der Funktionen:

$$T : F \longrightarrow SH \rightharpoonup \mathbb{P}(SH).$$

Der schwierigste Teil in der Definition der Semantik
ist der Umgang mit Funktionsaufrufen.

7.4 Semantik von Funktionsaufrufen

Um

$$[\![v := f(\bar{a})]\!]_T \,(A, H)$$

zu bestimmen, durchlaufen wir alle Spezifikationen

$$\{P\}\ f(\bar{x})\ \{Q\} \quad \text{in } T(f) \quad (\text{also } [T(f)](P) = Q).$$

Jetzt rufen wir auf

$$\text{BI-ABD}(H * X,\ P\{\bar{x}/\bar{a}\} * Y).$$

Falls Bi-Abduktion failed, gehen wir zur nächsten Spezifikation.

Sonst finden wir Lösungen

↳ X = wird noch in H benötigt, um P herzustellen.

↳ Y = wird aus H nicht benötigt, um P herzustellen.

Jetzt fügen wir

$$(A * X,\ Q\{\bar{x}/\bar{a}\} * Y)$$

der Menge $\quad [\![v := f(\bar{a})]\!]_T \,(A, H) \quad$ hinzu.

Das ist eine vereinfachte Darstellung.

Es muss auch noch Renaming gemacht werden.

Das machen wir am Beispiel.

Beispiel:

Sei

$$\{\overbrace{x \mapsto m' * y \mapsto n'}^{P}\}\ swap(x,y)\ \{\overbrace{ret = 0 \wedge x \mapsto n' * y \mapsto m'}^{Q}\}$$

die einzige Spezifikation von swap in T.

(Beachte, dass freie Variablen für Adressen in der Definition von $\models \{A\}\, c\, \{B\}$ vom Allquantor über dem Stack behandelt werden.)

Wir bestimmen

$$[\![v := swap(x,y)]\!]_T\ (A, H)$$

mit

$$A := x = m \wedge y = n \wedge m \mapsto o$$

$$H := y = n \wedge x \mapsto y * z \mapsto o.$$

Dann liefert

$$BI\text{-}ABD(y = n \wedge \underline{x \mapsto y} * \underline{z \mapsto o} * X,\ \underline{x \mapsto m'} * \underline{\underline{y \mapsto n'}} * Y)$$

als Lösung

$$X = \underline{m' = y} \wedge \underline{y \mapsto p} \wedge \underline{\underline{n' = p}}$$

$$Y = \underline{z \mapsto o}$$

Wir haben also:

Frame $z \mapsto o$

Anti-Frame $y \mapsto p$

Instantiierung der Funktionsparameter: $m' = y \wedge n' = p$.

Damit erhalten wir

$$[\![v := swap(x,y)]\!]_T \,(A, H) =$$

$$\{ (\underbrace{\underbrace{x = m \land y = n \land m \mapsto o}_{A} * n \mapsto p}_{\text{neues } A} , \underbrace{\underbrace{v = o \land x \mapsto p * y \mapsto y}_{Q\{x/x,\, y/y\} + \text{Rename}} * \underbrace{z \mapsto 0}_{\text{Frame}}}_{\text{neues } H}) \}$$