Simple Loose Ownership Domains: Ein Typsystem zur Kapselung von Objekten

Jan Schäfer und Arnd Poetzsch-Heffter

1 Einleitung

Java und andere objekt-orientierte Programmiersprachen bieten Kapselung nur auf Attribut-Ebene an. So ist es in Java z. B. möglich den Zugriff auf Klassenattribute mit dem Schlüsselwort `private` auf Quelltext der gleichen Klasse zu beschränken. Dieser Ansatz ermöglicht es aber nur Werte von Attributen zu schützen. Objekte werden in Java durch Referenzen angesprochen. Um Objekte in Attributen zu speichern kann man nur die entsprechenden Referenzen ablegen, nicht jedoch die Objekte selbst. Dadurch ist es in Java nicht möglich Objekte zu kapseln. Um dies zu verdeutlichen ein kleines Beispiel. Es zeigt die Implementierung der `Class`-Klasse des JDK 1.1, die zu einer Sicherheitslücke geführt hat [6]:

```
public class Class {
   private Identity[] signers;
   public Identity[] getSigners() {
      return signers;
   }   ...
}
```

Das `signers`-Array enthält alle vertrauenswürdigen Identitäten. Obwohl das Attribut selbst als `private` deklariert wird, können beliebige Applets sich in das Array eintragen, da die `getSigners()`-Methode die Referenz auf das interne Array zurückgibt. Der Programmierer der Methode hatte vergessen eine Kopie des Arrays anzulegen. Das Beispiel verdeutlicht sehr gut das Problem. Objekte (wozu wir hier auch Arrays zählen) sind in der Regel veränderliche Datenstrukturen. Da in Java Objekte nur über Referenzen angesprochen werden, muss sichergesellt werden, dass Referenzen auf interne Objekte nicht nach außen gelangen können, da diese Objekte sonst direkt verändert werden können, ohne die Schnittstelle des eigentlichen *Besitzerobjektes* zu benutzen. Mit Java-eigenen Mitteln ist es nicht möglich, dies zu garantieren. Programmierer müssen dies selbst durch Programmierdisziplin sicherstellen.

2 Typsystem-Erweiterung

Um die Kapselung von Objekten zu garantieren haben wir eine Typsystem-Erweiterung für Java entwickelt. Diese Erweiterung ermöglicht es statisch, d. h. zur Übersetzungszeit, zu garantieren, dass bestimmte Objekte gekapselt sind. Unsere Arbeit basiert allgemein auf der Idee von *Ownership*-Typsystemen [3, 4, 2] und speziell auf den sog. *Ownership Domains* [1].

Jedes Objekt in unserem System wird eindeutig einem Objekt-Bereich (*domain*) zugeordnet. Jeder Objekt-Bereich gehört wiederum genau einem Objekt, dem *Besitzerobjekt*. Dadurch entsteht eine baumförmige Relation von Objekten und Bereichen, wobei ein spezieller globaler Bereich (*global* domain) die Wurzel darstellt. Jedes Objekt besitzt genau zwei Bereiche: Einen *Grenz*bereich (*boundary* domain) und einen *lokalen* Bereich (*local* domain). Objekte im lokalen Bereich eines Objektes X sind gekapselt und können von außen nicht direkt referenziert werden. Objekte im Grenzbereich von X können auf den lokalen Bereich zugreifen und sind gleichzeitig von Objekten außerhalb referenzierbar.

Um diese Eigenschaften statisch zu garantieren erweitern wir das Typsystem von Java um *Bereichsannotation*. Jeder Referenztyp, d. h. Klassen-, Interface- und Arraytyp, wird um eine zusätzliche Annotation erweitert, die eindeutig festlegt, in welchem Bereich sich die entsprechenden Objekte zur Laufzeit befinden. Eine Bereichsannotation besteht dabei aus zwei Teilen: Dem *Besitzerteil* und dem *Bereichsteil*. Der Besitzerteil legt eindeutig das Besitzerobjekt des Bereiches fest. Der Bereichsteil legt fest, ob es sich um den Grenzbereich oder den lokalen Bereich handelt. Die Typdeklaration `this.local List` zum Beispiel legt fest, dass Objekte vom Typ `List` sind und sich in der lokalen Domain des aktuellen `this`-Objektes befinden. Es gibt drei Möglichkeiten den Besitzer eines Bereichs zu deklarieren: `this`, `owner` und `x`. Falls der Besitzerteil weggelassen wird, wird `this` angenommen. `this` steht für das aktuelle Empfängerobjekt, `owner` steht für das Besitzerobjekt des Bereiches, in dem sich das aktuelle `this`-Objekt befindet und `x` kann eine `final` Variable oder ein `final` Feld sein. Für die Annotation des Bereichsteils, gibt es drei Schlüsselwörter: `local`, `boundary` und `same`. `local` steht für den lokalen Bereich, `boundary` für den Grenzbereich und `same` steht für den Bereich in dem sich das Besitzerobjekt selbst befindet. Zusätzlich gibt es die Bereichsannotation `global`, die für den globalen Bereich steht.

Neben diesen so genannten *präzisen* Annotation kennt unser System auch *unpräzise* (*loose*) Annotationen. Unpräzise Annotation repräsentieren eine *Menge* von möglichen Objektbereichen. D. h. sie abstrahieren von dem exakten Bereich. Bei einer

```
public class LinkedList<T> {
  local Node<T> head;
  void add(T o) {
    head = new local Node<T>(o,head);
  }
  boundary Iter<T> iter() {
    return new boundary Iter<T>(head);
  } }
public class Iter<T> {
  owner.local Node<T> cur;
  Iter(owner.local Node<T> head) { cur = head; }
  boolean hasNext() { return cur != null; }
  T next() {  T res = cur.data;
    cur = cur.next; return res;
  }
}
```

```
public class Node<T> {
  T data;  same Node<T> next;
  Node(T d, same Node<T> n) {
    data = d; next = n;
  }
}
public class Main {  ...
  final local LinkedList<local Object> list;
  list = new LinkedList<local Object>();
  list.add(new local Object());
  list.boundary Iter<local Object> it;
  it = list.iterator();
  local.boundary Iter<local Object> it2;
  it2 = it;
  local Object obj = it2.next();  ...
}
```

Abbildung 1: Einfach verkettete Liste mit Iterator

unpräzisen Bereichsannotation ist der Besitzerteil der Annotation wiederum eine Bereichsannotation. `this.local.boundary` steht z. B. für alle Grenzbereiche von Objekten aus dem lokalen Bereich des `this`-Objektes. Unpräzise Annotation ermöglichen Implementierungen, die allein mit präzisen Annotationen nicht möglich sind (siehe [5]).

3 Beispiel: Liste mit Iterator

Um das Typsystem zu verdeutlichen stellen wir hier eine Implementierung einer verketteten Liste mit Iterator dar (Abb. 1). Sie besteht aus drei Klassen, einer `LinkedList`-Klasse, einer `Node`-Klasse und einer `Iterator`-Klasse. Die Implementierung muss sicherstellen, dass keine `Node`-Instanzen von außen referenzierbar sind, ansonsten könnte die Liste beliebig verändert werden ohne die vorgesehen Schnittstelle zu verwenden. Unser System garantiert dies, wenn die `Node`-Instanzen im lokalen Bereich des entsprechenden `LinkedList`-Objektes liegen. Das `head`-Attribut der `LinkedList`-Klasse ist daher mit (`this.`)`local` annotiert. Jeder Knoten hat eine Referenz auf seinen Vorgängerknoten. Da dieser im gleichen Bereich liegt wie der Knoten selbst, wird das entsprechende `next`-Attribut mit `same` annotiert. Iterator-Objekte der Liste müssen auf die Knoten der Liste zugreifen können und gleichzeitig von Objekten außerhalb der Liste erreichbar sein. Deswegen liegen sie im Grenzbereich der Liste und werden entsprechend mit `boundary` annotiert. Das Attribut `cur`, das auf den aktuellen Knoten des Iterators verweist, ist mit `owner.local` annotiert, das dem lokalen Bereich der Liste entspricht. Alle drei Klassen sind mit `T` parametrisiert. Der Parameter steht dabei sowohl für den eigentlichen Typ der Elemente der Liste, als auch für den Objekt-Bereich, in dem sich die Elemente befinden. Der Quelltext der `Main`-Klasse zeigt die Benutzung der `LinkedList`-Klasse. Um auf den präzisen Grenzbereich zugreifen zu können wird die `list`-Variable mit `final` deklariert. Der Iterator selbst wird mit `list.boundary` annotiert (`it`). Unser System erlaubt aber auch eine unpräzise Annotation mit `local.boundary` (`it2`).

4 Zusammenfassung

Objekt-orientierte Programmiersprachen bieten keinen Mechanismus um Objekte zu kapseln. Wir haben eine Typsystemerweiterung für Java entwickelt, die Kapselung von Objekten statisch garantiert. Eine ausführlichere Beschreibung, sowie eine Formalisierung und einen Korrektheitsbeweis findet sich in [5].

Literatur

[1] Aldrich, J. ; Chambers, C. : Ownership Domains: Separating Aliasing Policy from Mechanism. In: *Proc. ECOOP'04* Bd. 3086, Springer-Verlag, Jun. 2004 (LNCS), S. 1–25

[2] Boyapati, C. ; Liskov, B. ; Shrira, L. : Ownership Types for Object Encapsulation. In: *Proc. POPL '03*, ACM Press, Jan. 2003, S. 213–223

[3] Clarke, D. ; Potter, J. ; Noble, J. : Ownership Types for Flexible Alias Protection. In: *Proc. OOPSLA '98*, ACM Press, Okt. 1998, S. 48–64

[4] Müller, P. ; Poetzsch-Heffter, A. : Universes: A Type System for Controlling Representation Exposure. In: Poetzsch-Heffter, A. (Hrsg.) ; Meyer, J. (Hrsg.) ; Fernuniversität Hagen (Veranst.): *Programmiersprachen und Grundlagen der Programmierung, Kolloquiumsband '99* Fernuniversität Hagen, 2000 (Informatik Berichte 263–1)

[5] Schäfer, J. ; Poetzsch-Heffter, A. : Simple Loose Ownership Domains / Fachbereich Informatik, TU Kaiserslautern. 2006 (348/06). – Forschungsbericht. – Abrufbar unter `http://softech.informatik.uni-kl.de/~janschaefer`

[6] Vitek, J. ; Bokowski, B. : Confined types in Java. In: *Software – Practice and Experience* 31 (2001), Nr. 6, S. 507–532