Non-null Types in an Object-Oriented Language

Seminararbeit
Prof. Dr. P. H. Schmitt
Institut für Institut für theoretische Informatik
Universität Karlsruhe

vorgelegt von
Saoussen Arfaoui

Betreuer: Dipl. Inform. Benjamin Weiss

Saoussen Arfaoui
Matrikelnummer: 1178083

Inhaltsverzeichnis

1 Einleitung **1**

2 Non-null-Types **3**

2.1 Notation und Anwendung 3

2.2 Vorteile 4

3 Probleme **6**

3.1 Objektkonstruktion 6

3.1.1 Problem 6

3.1.2 Lösung: Raw Typen und Subtypen 6

3.2 Array-Typen 8

3.2.1 Problem 8

3.2.2 Lösung 8

3.3 Value-Types 8

3.3.1 Problem 8

3.3.2 Lösung 9

4 Zusammenfassung **10**

Literaturverzeichnis **III**

Kapitel 1

Einleitung

Während der Entwicklung eines Software Produktes, versucht der Entwickler alle möglichen Bugs zu beseitigen. Allerdings gibt es Fehler die erst zur Laufzeit auftauchen. Spät auftauchende Fehler verletzen aber die Robustheit einer Software und können zu massiven Verluste führen. Das referenzieren von Null in einer Objekt orientierten Sprache wie Java oder C#, ist eine Fehlerquelle die ein Programm zum Abstürzen bringt. Beispiel:

```
1   class A {
2      String a;
3      Public A (String a) {
4          this.a = a;
5          this.m (25);
6      }
7  void m (int x) { }
8  }
9  class B extends A {
10     String b;
11     public B (String b, String a){
12       super(a);
13       this.b = b;
14     }
15 @Override
16  void m (int x){
17      this.b ;
18    }
19 }
```

im obigen Beispiel wird bei der Konstruktion von Objekt B, zunächst das Objekt A dann das Objekt B konstruiert. Das Problem taucht bei der Konstruktion des Objektes A auf, wo die Methode m von der Klasse B aufgerufen wird(Polymorphie).In dieser Methode wird auf dem Attribut b zugegrieffen,ohne initialisiert zu werden, weil der konstruktor von B noch nicht aufgerufen ist.

Eine Lösung dafür wäre den Zugriff auf ein Objekt, der gerade aufgebaut wird, zu verbieten, bis das ganze Objekt erzeugt wird. und das Typsystem der

Programmiersprache so zu erweitern, dass man zwischen null und non-null Referenzen unterscheiden kann. Diese Mechanismen existiert schon in manchen Programmiersprachen, wie ML oder Haskel. Diese Programmiersprachen erlauben den Zugriff auf einem Objekt erst wenn er komplett erzeugt wird, im gegensatz zu Java oder C#.

Kapitel 2

Non-null-Types

2.1 Notation und Anwendung

Für Referenz-Typen werden 2 Subtypen, für Klassen und Interfaces unterschieden. T^- steht für non null Referenz des Typs T, T^+ steht dagegen für possibly-null Referenz oder auch einfach T da T defaultmässig null referenzieren kann. Ein Objekt vom Typ T^+ bzw. T kann eine variable vom Typ T^+ oder auch vom Typ T^- zugewiessen bekommen, und somit ist T^- wieder ein Subtyp von T^+. (siehe Beispiel).Zu einem Objekt vom Typ T^- können wir nur ein Objekt vom Typ T^- zuweisen d.h es kann nur non null sein.

Beispiel:

T^- t = new T () // ein non Null Objekt erzeugen

T^+ n = t // ist erlaubt (ein possibly null type kann auch non null sein)

... if(n!=null) t=n; // n hat den Typ T^-

int x = t.f // t muss vom Typ non null sein damit der Zugriff auf f

mittels der . Operator problemlos läuft.

Die Sprache muss so typisiert werden dass Zugriffe auf null referenzen vom Kompiler entdeckt werden und keine Ausnahmen zur Laufzeit zu verursachen. z.B anstatt eine Nullpointerexception zur Laufzeit zu werfen, wird der Compiler auf einen expliziten Type cast hinweisen oder einen Objekt vom Typ T^- erwarten. Z.B die Parameter bzw. die Ausgabetypen einer Methode werden vom Typ T^- deklariert bzw. zurückgegeben, so dass ein Zugriff mittels den Operator ".ïmmer ohne null referenz Exception durchgeführt wird.

In Bezug auf die Vererbungshierarchie, wenn S eine Unterklasse von T ist oder wenn T ein super-Interface von S ist , dann ist S^+ ein unter-Typ von T^+ ($S^+ <: T^+$) bzw. S^- ein unter-Typ von T^-($S^- <: T^-$). Possibly-null bedeutetet es könnte eine null wie auch eine

non-null Referenziert werden d.h ein T^- Typ ist zu einem T^+ Typ kompatible und somit ist T^- ein unter-Typ von T^+ das ergibt dann S^- ein unter-Typ von T^+ $((S^- <: T^+))$(siehe Abbildung 2.1)(siehe [2]).

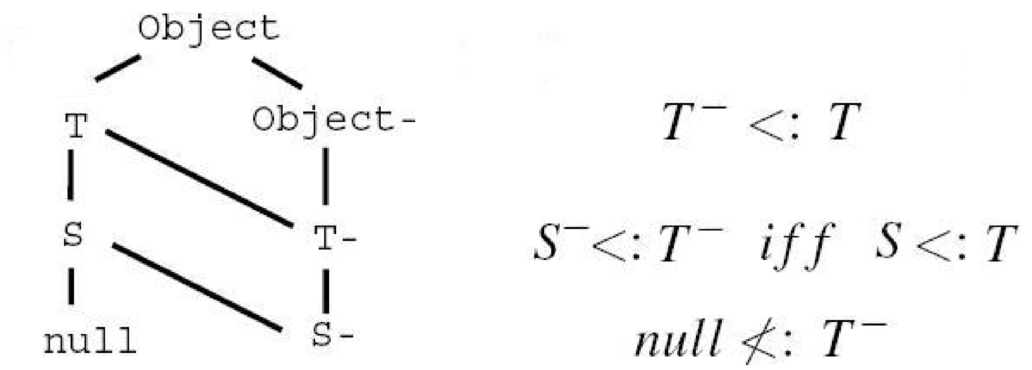

Abbildung 2.1: Beziehung zwischen den Typen.

Für C# und Java wird der Ausdrücke e Instanceof T nicht zu e instanceof T^+, oder e instanceof T^- erweitert da der non-nullity Test implizit damit geprüft wird. wobei mit instanceof wird der Ausdruck e geprüft ob er vom Typ T, der nicht null ist oder nicht.

2.2 Vorteile

die Vorteile der Non-null-Types in eine Objekt orientierte Sprache:

- Dokumentation Input-Parameter, Output- Parameter und Rückgabewerte der Methode: die Aufrufer der Methode wissen dann wann die Methode ein non-null Argument erfordert und wann ein non-null wert zurückgibt.
- Statische Überprüfung der Objektinvarianten: Objektinvarianten, die non-null Referenzen erfordern können deklariert und zur Kompilierzeit überprüft werden.
- Präzise Fehlererkennung: der Fehler, der auftaucht wenn eine null Referenz benutzt wird während das Programm eine nonnull Referenz erwartet, wird an der Stelle detektiert wo das Problem aufgetreten ist.
- Optimierung der Leistung: Keine Notwendigkeit zu prüfen, ob das Objekt Null referenziert wenn es schon vom Typ non-null ist.
- Wenige null-Referenz Ausnahmen: Verringerung der Anzahl NullReferenceExceptions durch die Benutzung von non-null Referenzen .

- Grundlage anderer Prüfer: es ist einfacher Programmprüfer zu schreiben, die die null Referenzen nicht betrachten müssen, was die Quelle der meisten Warnungen darstellt.

Wie gesehen sind also non-null-Types eine sehr praktische und hilfreiche Lösung der null-Referenz Problem. Im nächsten Kapitel wird der Einsatz der Non-Null-Types im Objektkonstruktion, Arrays und Value-Types vorgestellt .

Kapitel 3

Probleme

3.1 Objektkonstruktion

3.1.1 Problem

Das Problem taucht auf, wenn in einer Klasse C, während der Objektkonstruktion, auf eine Instanz Variable f dieser Klasse zugegriffen wird. Wenn f als non- null Type T^- deklariert ist, und c vom Typ C^- ist, this.f kann Null zurückgeben, weil This auf das Objekt während der Konstruktion zugreift, und Java und C# verbieten nicht diesen Zugriff. Betrachten wir noch mal das Einleitungs-Beispiel; alle Attribute der Klasse A sind fertig initialisiert, sobald der Attribute a im Kostruktor initialisiert wird(this.a = a), und damit sind alle super Klassen Attribute auch initialisiert. Wir wiessen aber noch nicht ob die Attribute der Unterklassen von A fertig initialisiert sind oder nicht. Der Aufruf von der Methode m im Konstruktor A(this.m(25)) liefert eigentlich nicht ein Objekt vom typ A und ist deswegen virtual, weil virtuelle Afruffe implizit über den Zustand der Unterklasse, die noch nicht initialisiert ist informieren. Das Problem ist nicht nur auf den Fall von this begrenzt, es könnte auch auftretten wenn es auf f über eine andere Instanz x zugegriffen wird. der aufruf x.f kann null sein sogar wenn f als non-null T^- deklariert ist.

3.1.2 Lösung: Raw Typen und Subtypen

Die Lösung wäre: für jedes Referenztyp T, ein neues Typ T^{raw^-} einzufügen. Wobei T^{raw^-}, steht für die teilwiese initialisierte Objekte der Klasse T oder der Unterklasse davon. T^{raw^-} bezeichnet eigentlich dieselbe Struktur wie T^- außer dass die Attribut Zugriffe über diese variable null ergeben dürfen. Bei einem lese Zugriff, der Aufruf c.f kann null liefern wenn c vom Typ C^{raw^-} ist und f vom Typ T^-. Bei einem schreibe Zugriff dagegen, erfordert ein c.f Aufruf ein non-null-Typ also ein C^- Typ. Beispiel:

```
 1  class A {
 2      [NotNull]
 3      String a;
 4        Public A ([NotNull]String a) {
 5              this.a = a;
 6              this.m (25);
 7        }
 8  [RAW]
 9  void m (int x) { }
10  }
11  class B extends A {
12      [NotNull]
13      String b;
14      public B ([NotNull] String b, [NotNull] String a){
15        super(a);
16        this.b = b;
17      }
18  @Override
19  [RAW]
20   void m (int x){
21        this.b ;
22     }
23  }
```

Wenn es eine Vererbungshierarchie zwischen mehrere Klassen gibt, und wenn die Erzeugung der Objekt einer Unterklasse von Attribute der oberen Klassen abhängt, so wird das Objekt in Klass. Frames aufgeschlüsselt. Jedes Rahm(frame) representiert eine Oberklasse. Für jedes T^{raw^-} Objekt, ist $T^{raw(S)^-}$ der Typ des gerade fertig initialisiert Teil-Objektes. Wenn bei der Objekt-Konstruktion den typ $T^{raw(T)^-}$ erreicht wird dann ist die erzeugeung fertig durch geführt, weil $T^{raw(T)^-}$ und T^- sind gleich. . Allerdings müsste der letzte Konstruktor dafür sorgen dass, der neu gebautes Objekt vom Typ T^{raw^-} in Typ T^- zu casten. Einmal die Konstruktion fertig ist, müsste jedes non-null Attribut der Klasse, ein non-null Wert zugewiessen bekommen, deswegen müssen auch alle Pfade im Konstruktor, alle non-null Attribute mit non-null Werte initialisieren. Wenn S eine Unterklasse von T dann ist S^{raw^+} eine Unterklasse von T^{raw^+} und S^{raw^-} eine Unterklasse von T^{raw^-}. Insbesondere wenn S eine Unterklasse von R ist dann ist $T^{raw(S)^-}$ wieder eine Unterklasse von $T^{raw(R)^-}$, d.h der letzte der in die initialisierungsreihe kommt, ist der, der den kleinsten Teil von seinem Konstruktor initialisiert hat (siehe [1]). Abschließend, gibt es Den T^{raw^+} type für possibly-null typen für teilweise initialisierte Objekte aber der ist praktisch nicht sehr nützlich wie T^{raw^-}. Darüberhinaus für jeden Typen T gilt T^{raw^-} unter Typ von T^{raw^+} und T^+ unter Typ von T^{raw^+}.

3.2 Array-Typen

3.2.1 Problem

Ein Array kann null oder non-null typen als Elemente haben und kann selber als null bzw. non-null deklariert sein. Es gibt 4 veschiedene Array-Typen von jedem Referenz Typ T:

- $T^{-}[]^{-}$: non-null Array mit non-null Elemente
- $T^{+}[]^{-}$: non-null Array mit possibly null Elemente
- $T^{-}[]^{+}$: possibly-null Array mit null Elemente
- $T^{+}[]^{+}$: possibly-null Array mit possibly-null Elemente

Das selbe Problem wie bei der Objektkonstruktion gibt es auch bei der Arraykonstruktion, insbesondere taucht das Problem bei der Erzeugung von einem Array mit Elemente vom Typ non-null auf.

3.2.2 Lösung

Ähnlich wie bei der Objektkonstruktion ist hier auch ein Raw Typ erforderlich. Der Ausdruck new T^{-}[n] wird zu einem Array vom Typ $T^{-}[]^{raw^{-}}$. Wobei $T^{-}[]^{raw^{-}}$ für die teilwiese initialisierte Array vom Typ T^{-} steht. Es gelten hier auch dieselben lese bzw. schreib regeln wie bei der Objektkonstruktion; es kann beim lesen ein null zurückgegeben wird, obwohl die Array vom Typ $T^{-}[]^{-}$. Das schreiben in einem Array von dem selben Typ erfordert aber ein non-null Typ. Jedoch ist einen expliziten Typcast vom Typ $T^{-}[]^{raw^{-}}$ zu den Typ $T^{-}[]^{-}$ erforderlich, da der Compiler nicht erkennt wann die Array Konstruktion fertig ist. Ein typisches Programm-Stück für Array Initialisierung ist:

$T^{-}[]^{raw^{-}}$ tmp = new T^{-}[n]; tmp Elemente initializieren

$T^{-}[]^{-}$ array = ($T^{-}[]^{-}$)tmp;

3.3 Value-Types

3.3.1 Problem

Value-Types sind mittels struct in C# deklariert. Das Problem tauch auf bei der Konstruktion von einem struct Konstruktor seit dem es erlaubt wurde, non-null Types zu

benutzen. Der struct standard Konstruktor, initialisiert alle seine Attribute mit default Werten, und kann nicht überschrieben werden.

3.3.2 Lösung

Hier ist die gleiche Lösungsidee wie für Objekte nicht anwendbar aber ein raw Typ ist aufjedenfall erforderlich. Um das ganze zu vereinfachen wir unterscheiden zwischen istructs und structs. Ein istruct ist ein struct bei dem der standard konstruktor nicht ausreichend ist.Ein Struct ist ein istruct wenn er ein non-null deklariertes attribut oder ein istruct hat. S^{raw} wird dann für die teilweise initialisierte Structs benutzt, und wird nur von einem istruct standard konstructor erzeugt. wenn der Attribut ein istruct vom Typ S ist dann kann ein lese zugriff ein S^{raw} Typ zurückgeben. Ein schreib zugriff dagegen erfordert ein S typ. Alle anderen lese zugriffe bleiben analog wie beim Objekt konstruktion.

Kapitel 4

Zusammenfassung

Im Rahmen dieser Seminararbeit wurde das neue Konzept Erweiterung der Sprachen, wie Java oder C# um non null types, vorgestellt. Mit Hilfe dieser Erweiterung wird die Anzahl der Laufzeitfehler verringert und somit wenig unerwartete NullReferenzexceptions. Non-null Types sind für die vermeidung des referenzierens von null gedacht. Wenn ein Attribut einer Klasse non-null ist, bekommt er den Typ T^- sonst den Typ T^+. Raw^-Types sind hinzugefügt damit es kein Konflikt bei der Objektkonstruktion auftritt, Jedes Objekt unter Konstruktion ist vom typ T^{raw^-} bis er komplett initialisiert wird erst dann bekommt er den Typ T^-. Das Lesezugriff auf einem Objekt während seine Konstruktion ist erlaubt. Ein schreibzugriff ist erst möglich wenn das Objekt Komplett erzeugt ist. Eine Implementierung der Non-null Types existiert, ist aber noch nicht vollständig.

Literaturverzeichnis

[1] K.Rustan M.Leino Manuel Fähndrich. *Declaring and Checking Non-null Types in an Object-Oriented Language.* Microsoft Research One Microsoft Way remond, WA 98052,USA.

[2] Görel Hedin Torbjörn Ekman. *Pluggable checking and inferencing of non-null types for Java.* Journal of Object Technology 6(9):455-475,2007. Special Issue: TOOLS EUROPE 2007.