Implementierung des Universe-Typsystem in ESC/Java 2

Dirk Wellenzohn

Semester Projekt Report

Software Component Technology Group
Department of Computer Science
ETH Zurich

http://sct.inf.ethz.ch/

SS 2005

Betreut von:
Dipl.-Ing. Werner M. Dietl
Prof. Dr. Peter Müller

Software Component Technology Group
inf | Informatik
Computer Science

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Zusammenfassung

Das Ziel dieser Semesterarbeit ist die Implementierung des Universe-Typsystem in ESC/Java 2. Das erfordert die Analyse von ESC/Java 2 und das Auffinden von Möglichkeiten das Parsing und das Typchecking zu erweitern.

Kapitel 1 erklärt das Universe-Typsystem und dessen Regeln. Kapitel 2 beschreibt kurz die Architektur und Funktionsweise von ESC/Java 2. Kapitel 3 zeigt die gewählte Implementation und begründet Designentscheidungen. In Kapitel 4 werden ein Fazit gezogen und offene Punkte angesprochen.

Inhaltsverzeichnis

1 Universe-Typsystem **7**
- 1.1 Einleitung . . . 7
- 1.2 Typen . . . 7
- 1.3 Typ-Kombinator . . . 8
- 1.4 Subtyping . . . 8
- 1.5 Arrays . . . 9
- 1.6 Regeln . . . 9
  - 1.6.1 Instanzfelder . . . 9
  - 1.6.2 Statische Felder . . . 9
  - 1.6.3 Objekterzeugung . . . 9
  - 1.6.4 Instanzmethoden . . . 10
  - 1.6.5 Aufruf von Instanzmethoden . . . 10
  - 1.6.6 Statische Methoden . . . 10
  - 1.6.7 Statische Methodenaufrufe . . . 10
  - 1.6.8 Pure Methoden . . . 10
  - 1.6.9 Überschreiben und Überladen . . . 10
  - 1.6.10 Konstruktoren . . . 10
  - 1.6.11 Exceptions . . . 10
  - 1.6.12 Innere Klassen . . . 10

2 ESC/Java 2 **11**
- 2.1 Einleitung . . . 11
- 2.2 Arbeitsweise . . . 11
- 2.3 Architektur . . . 11
- 2.4 Lexikographische Analyse . . . 12
- 2.5 Parsing . . . 13
- 2.6 Abstract-Syntax-Tree . . . 13
- 2.7 Typchecking . . . 13
- 2.8 Vorbereitung auf das Universe-Typsystem . . . 14

3 Implementierung des Universe-Typsystem in ESC/Java 2 **17**
- 3.1 Design-Entscheidungen . . . 17
  - 3.1.1 Ort der Implementierung . . . 17
  - 3.1.2 Art der Repräsentation . . . 17
- 3.2 Kommandozeilenparameter . . . 18
- 3.3 Details der gewählten Implementation . . . 18
  - 3.3.1 javafe.Options: . . . 18
  - 3.3.2 javafe.parser.TagConstants: . . . 18
  - 3.3.3 javafe.parser.ParseUtil: . . . 19
  - 3.3.4 javafe.parser.ParseExpr: . . . 22
  - 3.3.5 javafe.parser.ParseStmt: . . . 23
  - 3.3.6 javafe.parser.Parse: . . . 23

3.3.7 javafe.tc.FlowInsensiveChecks: . . . 24
3.3.8 escjava.ast.TagConstants: . . . 28
3.3.9 escjava.parser.EscPragmaParser: . . . 28
3.3.10 escjava.tc.FlowInsensitiveChecks: . . . 29
3.3.11 escjava.RefinementSequence: . . . 29

4 Fazit **31**
4.1 Erfahrungen . . . 31
4.2 Kritik am gewählten Ansatz . . . 31
4.3 Offene Punkte . . . 32
4.4 Testfälle . . . 32

Kapitel 1

Universe-Typsystem

1.1 Einleitung

Dieses Kapitel ist eine Übersetzung der „Universe Type System – Quick-Reference“ [1]. Das Universe-Typsystem [2] wurde entwickelt, um Aliasing und Abhängigkeiten in objektorientierten Sprachen zu kontrollieren. Es ist ein Ownership-Typsystem. Dazu wird der Objekt-Store hierarchisch in sogenannte Universen unterteilt. Ein Universum ist eine Menge von Objekten, welche den gleichen Besitzer (owner) haben. Jedes Objekt hat maximal einen Besitzer und alle Objekte in einem Universum haben denselben Besitzer. Objekte ohne Besitzer sind im Root-Universum. Das Ziel ist es, Representation-Encapsulation zu erzwingen und Leaking zu verhindern, ohne dabei Aliasing komplett zu verbieten.

Normale read-write Referenzen sind nur zwischen Objekten im gleichen Universum oder zwischen Besitzer und Objekten in dessen Universum erlaubt. Dadurch wird modulare Verifikation möglich. Alle anderen Referenzen sind readonly und erlauben keine Modifikation der referenzierten Objekte, weder durch Zuweisung an ein Feld noch durch Aufruf einer nicht-puren Methode. Die readonly-Referenzen sind transitiv, es ist also nicht möglich durch eine readonly-Referenz eine read-write-Referenz zu erhalten.

1.2 Typen

Um die Referenzen zu klassifizieren wird ein erweitertes Typsystem benutzt. Dazu werden drei neue Schlüsselwörter eingeführt. Da Aliasing nur bei Referenztypen auftritt, wird das Universe-Typsystem nur auf Referenztypen angewandt.

peer: bezeichnet eine Referenz im gleichen Universum und wird implizit als Standard genommen, solange nichts anderes vermerkt ist.
rep: bezeichnet eine Referenz von einem Objekt in ein Universum, das es besitzt.
readonly: bezeichnet eine readonly-Referenz in ein beliebiges Universum.

Diese Schlüsselwörter werden Universe-Typ-Modifier genannt (oder kurz Universe-Modifier) und sie können vor den Standard-Java-Typen stehen. Für Referenztypen und Arrays primitiver Typen ist nur ein Universe-Modifier erlaubt, bei Arrays von Referenztypen hingegen noch ein zweiter für die Elemente.

Wir benennen die Referenzen bezüglich ihres (ersten) Universe-Modifier entweder peer-, rep- oder readonly-Typ.

Diese Schlüsselwörter können direkt im Javacode benutzt werden, aber dann kann der Code nur mit dem MultiJava/JML-Compiler kompiliert werden. Alternativ können sie innerhalb von JML-Kommentaren benutzt werden. Innerhalb von JML-Kommentaren können auch \peer, \rep und \readonly benutzt werden, welche von nicht auf das Universe-Typsystem vorbereiteten JML-Tools ignoriert werden.

Beispiele:

```
peer T a;              // nur mit ESC/Java 2 und MultiJava benutzbar
/*@ peer @*/ T b;     // als JML-Spezifikation
/*@ \peer @*/ T c;    // als JML-Spezifikation, alternativ
```

JML erlaubt das Markieren von Methoden als pur, wenn sie keine Seiteneffekte haben. Auf readonly-Referenzen können nur pure Methoden ausgeführt werden. Im Gegensatz zu den Universe-Typ-Modifier ist der JML-Modifier pure nur in JML-Kommentaren erlaubt (anders als bei Multijava).

1.3 Typ-Kombinator

Um den Universe-Typ eines Objektes, welches von x.f referenziert wird, zu bestimmen – also den Typ des Feldzugriffs x.f – müssen beide Modifier betrachtet werden, der von x und der von f.

- Wenn beide peer sind, dann wissen wir, (a) dass das von x referenzierte Objekt den gleichen Besitzer hat wie this und, (b) dass x.f den gleichen Besitzer hat wie x und damit wie this. Deshalb hat x.f auch den Universe-Typ peer.
- Wenn der Typ von f rep ist, dann hat this.f den Universe-Typ rep, weil das durch this.f referenzierte Objekt this als Besitzer hat.
- Wenn der Typ von x rep und derjenige von f peer ist, dann hat x.f den Universe-Typ rep, weil (a) das durch x referenzierte Objekt ist im Besitz von this, und (b) das durch x.f referenzierte Objekt hat den gleichen Besitzer wie x, also this.
- In allen anderen Fällen können wir nicht bestimmen, ob das durch x.f referenzierte Objekt this als Besitzer oder den gleichen Besitzer wie this hat. Deshalb setzen wir in diesen Fällen den Universe-Typ von x.f auf readonly.

Die gleichen Regeln werden auch für Methodenaufrufe benutzt und können als ein Typ-Kombinator ausgedrückt werden, welcher zwei Universe-Typen nimmt und den resultierenden Typ zurück gibt. Dieser Kombinator wird benutzt, um die Universe-Typen von Feldzugriffen, Arrayelementzugriffen, Methodenrückgabewerten und Methodenparametern zu bestimmen.

Der Typ-Kombinator ist definiert durch die folgende Tabelle (erstes Argument: Zeile, zweites Argument: Spalte):

*	peer	rep	readonly
peer	peer	readonly	readonly
rep	rep	readonly	readonly
readonly	readonly	readonly	readonly

peer * rep ergibt eine rep Referenz, wenn der erste Parameter this ist.

1.4 Subtyping

Die Subtyp-Relation auf Universe-Typen folgt der Subtyp-Relation in Java: zwei peer-, rep- oder readonly-Typen sind Subtypen, wenn die entsprechenden Javaklassen oder Interfaces Subtypen in Java sind. Zusätzlich ist jeder peer- und rep-Typ ein Subtyp des readonly-Typ mit dem selben Klassen-, Interface- oder Arrayelement-Typ.

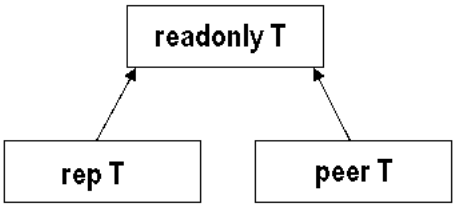

Abbildung 1.1: Typhierarchie der Universe-Typen

1.5 Arrays

- Arrays von Referenztypen brauchen zwei Universe-Modifier: einen für das Arrayobjekt und einen für die Elemente. Ist nur ein Modifier angegeben, wird er für die Elemente benutzt und das Arrayobjekt ist peer.
- Arrays von primitiven Typen brauchen nur einen Universe-Modifier, den für das Arrayobjekt.
- Arrayelementzugriffe werden wie Feldzugriffe behandelt, d.h. der Universe-Typ-Kombinator wird benutzt.
- Der Universe-Modifier für die Arrayelemente darf nicht rep sein.

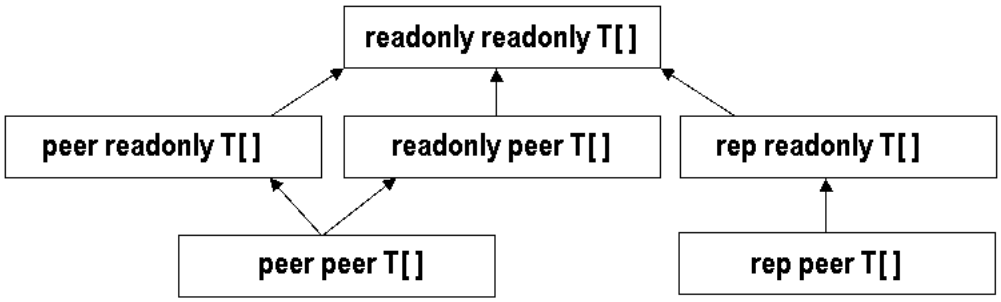

Abbildung 1.2: Typhierarchie für Arrays

1.6 Regeln

1.6.1 Instanzfelder

- Wird ein Feld mit Modifier rep von einem Target verschieden von this gelesen, dann hat der Ausdruck den Typ readonly (siehe 1.3 Universe-Typ-Kombinator).

1.6.2 Statische Felder

- Statische Felder können nicht rep sein, da sie kein Target haben.
- Eigentlich sollten alle statischen Felder readonly sein. Um die Benutzung von existierendem Code zu vereinfachen, wird jedoch peer als Standard benutzt und eine Warnung ausgegeben.

1.6.3 Objekterzeugung

- Nur peer und rep sind erlaubt für New-Ausdrücke, weil jedes Objekt einen Besitzer braucht und readoly keinen definieren würde.

1.6.4 Instanzmethoden

- Die This-Referenz gilt als peer-Typ.
- Wenn einer der Parameter rep ist, dann kann die Methode nur auf dem Target this aufgerufen werden.

1.6.5 Aufruf von Instanzmethoden

- Auf readonly-Referenzen sind nur pure Methoden aufrufbar.
- Auf den Rückgabewert und die formalen Parameter wird der Universe-Typ-Kombinator angewandt (bezüglich Target).

1.6.6 Statische Methoden

- Statische Methoden können keine rep-Typen benutzen, weder in ihrer Signatur noch als lokale Variablen.

1.6.7 Statische Methodenaufrufe

- Statische Methoden können als T.m() relativ zum aktuellen Kontext aufgerufen werden.
- Es ist auch möglich, die statischen Methoden als peer T.m() und als rep T.m() aufzurufen. Dies ist aber noch nicht implementiert. Alle statischen Aufrufe werden als peer T.m() interpretiert (siehe 4.3 offene Punkte).

1.6.8 Pure Methoden

- Alle Parameter von puren Methode müssen implizit oder explizit readonly sein.
- Es gelten die JML-Regeln für pure Methoden.

1.6.9 Überschreiben und Überladen

- Das Überladen von Methoden ist verboten, wenn sie sich nur in den Universe-Typen der Parameter unterscheiden.
- Pure Methoden können nur von puren Methoden überschrieben werden.

1.6.10 Konstruktoren

- Konstruktoren dürfen keine rep-Parameter haben, denn das neu erstellte Objekt kann nicht Besitzer des übergebenen Arguments sein.

1.6.11 Exceptions

- Der Typ des Throw-Ausdrucks ist readonly.
- Der Typ des Catch-Ausdrucks ist implizit oder explizit readonly.
- Für die Throws-Deklaration kann kein Universe-Modifier angeben werden. Der Typ der Throws-Deklaration ist immer readonly.

1.6.12 Innere Klassen

- Referenzen zu äusseren Instanzen sind readonly.

Kapitel 2

ESC/Java 2

2.1 Einleitung

ESC steht für „extended static checking“. „Static checking“, weil das Programm nicht laufen gelassen wird für die Checks und „extended“ (=erweitert), weil mehr Fehler entdeckt werden als bei herkömmlichen statischen Checks wie Typchecking. ESC/Java wurde ursprünglich am Systems Research Center von DEC (später Compaq, heute HP) entwickelt, wo man durch die Entwicklung von ESC/Modula-3 bereits Erfahrung auf diesem Gebiet hatte. ESC/Java nutzte eine eigenen Annotationssprache.

ESC/Java 2 wurde vor allem von David Cok und Joe Kiniry entwickelt, nachdem der Sourcecode von ESC/Java unter einer „relativ“ freien Lizenz freigegeben wurde. Sie integrierten Java 1.4 Features, änderten die Annotationssprache zu JML und erweiterten die statischen Checks. Wie die Entwicklung von JML ist auch ESC/Java 2 ein noch laufendes Projekt.

ESC/Java 2 arbeitet vollkommen automatisiert und eignet sich hervorragend, um mögliche Laufzeitfehler zu finden. Durch die Benutzung von JML-Spezifikationen können auch komplexere Dinge wie Vor- und Nachbedingungen oder Invarianten geprüft werden.

2.2 Arbeitsweise

In einer ersten Phase arbeitet ESC/Java 2 wie ein Compiler. Der Sourcecode (inklusive JML-Spezifikationen) wird geladen, geparst und auf Typsicherheit geprüft. Danach wird für jede Klasse ein Prädikat erstellt, welches Informationen (zu Typen von Feldern und JML-Spezifikationen) über diese Klasse enthält. Dann erfolgt eine Transformation der Methoden in Guarded-Commands, aus welchen schwächste Vorbedingungen und stärkste Nachbedingungen für die Methoden berechnet werden. Die zu prüfenden Eigenschaften werden in einer Verification-Condition zusammengefasst. Diese wird, unter der Annahme, dass das Prädikat der Klasse erfüllt ist, mit dem Theorembeweiser Simplify zu beweisen versucht. Wird ein Gegenbeispiel gefunden, so wird es in eine Fehlermeldung transformiert.

Dieses Vorgehen ist nicht „complete“, da es sein kann, dass Warnungen ausgegeben werden, weil ein Beweis in der gegebenen Zeit nicht gefunden wurde, obwohl er existieren würde. ESC/Java 2 ist zudem nicht „sound“, weil Loops so behandelt werden, als ob sie nur einmal ausgeführt würden. Zudem ist es erlaubt, dem Beweiser Hilfen in Form von Annahmen zu geben, welche dann nicht geprüft werden und damit falsch sein könnten.

2.3 Architektur

ESC/Java2 besteht aus zwei Teilen. Der erste Teil ist das Javafrontend Javafe, welches Javacode parst und typchecked. Dieses Frontend ist sehr flexibel, so dass es als Grundlage für verschiedenste

Javacode-verarbeitende Anwendungen benutzt werden kann. Indem man von Klassen aus Javafe erbt, kann man einen Grossteil des Verhaltens von Javafe durch überschreiben von Methoden ändern. Durch Factory-Methoden in javafe.FrontEndTool können alle wichtigen Komponenten zentral ausgetauscht werden. Javafe ist bereits darauf vorbereitet, Annotationen in Kommentaren zu verarbeiten (siehe 2.4).

Der zweite Teil ist Escjava, welcher Javafe um die Fähigkeit erweitert, JML zu verarbeiten und die komplexen statischen Checks durchzuführen. Dazu werden in escjava.Main gewisse Factory-Methoden (z.B. makeTypeCheck) aus javafe.FrontEndTool überschrieben. Escjava liefert die Implementation des Pragmaparsers, welcher vom Javafe-Lexer aufgerufen wird, um JML-Spezifikationen zu parsen. Die von escjava.Main gesetzte TypeCheck-Instanz kann auch die JML-Spezifikationen testen. Durch überschreiben von handleTD wird das Übersetzen in Guarded-Commands und schliesslich das Beweisen gestartet (siehe Methode processRoutineDecl in escjava.Main).

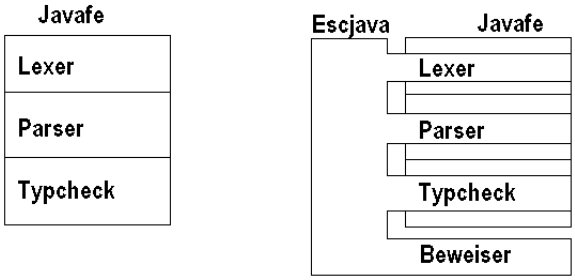

Abbildung 2.1: Architektur von Javafe und Escjava

2.4 Lexikographische Analyse

Ein javafe.parser.Lex Objekt generiert eine Sequenz von javafe.parser.Token. Diese Konversion folgt den Regeln von Kapitel drei der Java Language Specification. Bevor ein neu-kreiertes Lex-Objekt benutzt werden kann, muss die restart-Methode aufgerufen werden.

Die Methode getNextToken liefert das nächste Token, immer genau eines, und setzt den Lexer vor das nächste Token. Leerschläge werden verworfen und Kommentare speziell verarbeitet (siehe nächsten Abschnitt). Aus Effizienzgründen ist der Lexer selbst eine Subklasse von Token und getNextToken setzt als Seiteneffekt die Tokenfelder von this. Diese Tokenfelder sind ttype (ein Integer welcher den Typ des Tokens kodiert), startingLoc (Ort des erste Charakters), identifierVal (der Identifier, falls das Token einer ist) und auxVal (Wert, falls das Token ein Literal ist). Die Integerwerte für ttype, welche die Token kodieren, sind in der Klasse javafe.parser.TagConstants definiert. Ebenfalls wichtig ist die Methode lookahead, welche einen Integer als Parameter hat, der angibt wie weit voraus geschaut wird, ohne die Tokenfelder von this zu verändern.

Wird ein Kommentar angetroffen, wechselt der Lexer seinen Modus und ruft von getNextToken aus den Pragmaparser auf. Das Interface javafe.parser.PragmaParser definiert die Methode getNextPragma, welche den Kommentar parst und die Felder des mitgegebenen Token entsprechend füllt. Der bool'sche Rückgabewert zeigt dem Lexer, ob das nächste Token auch zu einer Annotation gehört oder ob der Lexer die Token wieder selbst scannen muss. Pragma ist die noch aus ESC/Java stammende Bezeichnung für JML-Spezifikationen. In Javafe sind 4 Arten von Pragmas vorgesehen: LexicalPragma, ModifierPragma, TypeDeclElemPragma und StmtPragma. Das Tokenfeld ttype muss mit einem der 4 entsprechenden Werte aus javafe.parser.TagConstants belegt werden und auxVal mit einer Instanz der entsprechenden Klasse (oder einer Subklasse davon). So kann Javafe den Parsebaum aufbauen, ohne JML-Spezifikationen kennen zu müssen. Die für ESC/Java 2 benutzte Implementation des Pragmaparserinterface ist escjava.parser.PragmaParser. Die Universe-Modifier gelten als ModifierPragmas.

2.5 Parsing

Eine Instanz der Klasse javafe.parser.Parse ist für das Parsing zuständig. Wie der Lexer ist auch der Parser ohne einen so genannten Parser-Generator erstellt worden. Zur Strukturierung ist das Parsing auf mehrere Klassen verteilt, welche jeweils Subklassen voneinander sind (siehe Abbildung 2.2).

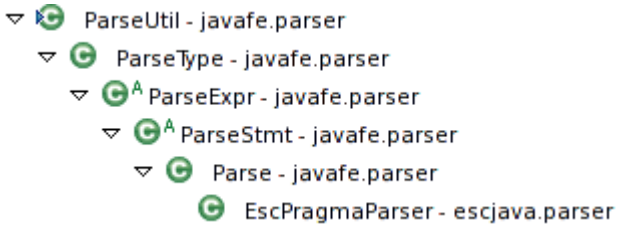

Abbildung 2.2: Klassenhierarchie der Parser-Klassen

Alle Parsingmethoden nehmen den Lexer als Argument und betrachten das aktuelle Token. Die Methodennamen verraten den Zweck der Methoden (z.B. parseSwitchStmt) sofort und zu Beginn steht jeweils die von ihr erkannte grammatikalische Produktion. Nun wird das aktuelle Token (und vielleicht per lookahead auch die nächsten) angeschaut, um zu sehen, ob die grammatikalische Produktion, welche diese Methode repräsentiert, angewandt werden kann. Kann die Produktion nicht angewandt werden, bleibt der Lexer beim aktuellen Token. Erkennt die Methode, dass die Produktion anwendbar ist, werden die Token der Produktion verarbeitet und der Lexer steht am Ende beim ersten Token der nächsten Produktion. Viele der Methoden arbeiten mit Seiteneffekten.

Beispiel für Seiteneffekte:
parseTypeDeclTail ruft für jedes TypeDeclElement (Feld, Methode, Konstruktor, Initblock) die Methode parseTypeDeclElemIntoSeqTDE auf, welche alle TypeDeclElemente erstellt und in einen Vektor (this.TypeDeclElemVec) einfügt. Am Ende nimmt parseTypeDeclTail den Vektor um die neue Typdeklaration zu erstellen.

2.6 Abstract-Syntax-Tree

Das Parsing baut einen Syntaxbaum auf, welcher aus Knoten von javafe.ast.ASTNode oder Subtypen davon besteht. Die Namen der Klassen sprechen für sich selbst (siehe Abbildungen 2.3, 2.4, 2.5 und 2.6).

2.7 Typchecking

Die Klasse javafe.tc.TypeCheck stellt Hilfsmethoden für das Typchecking und die Factorymethode makeFlowInsensitiveChecks zur Verfügung. Die Klasse javafe.tc.FlowInsensitiveChecks bietet die Methoden an, welche schliesslich den Parsebaum überprüfen. In Escjava wird TypeCheck erweitert und die Factorymethode überschrieben, so dass sie eine escjava.tc.FlowInsensitiveChecks-Instanz zurückgibt. Die beiden FlowInsensitiveChecks-Klassen implementieren das eigentliche Typchecking, wobei die Escjava-Erweiterung die JML-Spezifikationen prüft.

Alle Referenztypen (ausser Arrays) im Code werden zuerst als Instanzen von javafe.ast.TypeName im Parsebaum gespeichert. In der Typcheckingphase werden diese durch Instanzen von javafe.tc.TypeSig bzw. escjava.tc.TypeSig ersetzt. Das Auflösen (d.h. die symbolischen Referenzen im Parsebaum mit den entsprechenden Deklarationen zu verbinden) und Prüfen von Typen

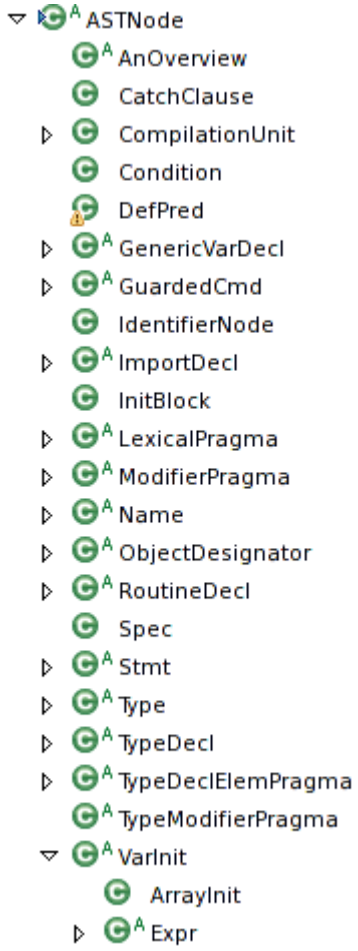

Abbildung 2.3: Klassenhierarchie des Abstract-Syntax-Tree (Ausschnitt)

ist durch die vielen gegenseitigen Abhängigkeiten sehr komplex. Die TypeSigs durchlaufen deshalb verschiedene Zustände, in denen sie schrittweise vervollständigt werden bis sie eine geprüfte Typdeklaration darstellen. Die TypeSig für jeden Typ ist einmalig, d.h. jede Referenz auf einen bestimmten Typ zeigt auf die eine TypeSig, die ihn repräsentiert. Dadurch kann Typgleichheit durch einen Referenzvergleich geprüft werden.

Auch in FlowInsensitiveChecks sind die Methodennamen gut gewählt (z.B. checkTypeDecl, checkStmt).

2.8 Vorbereitung auf das Universe-Typsystem

ESC/Java 2 war bereits auf die Verwendung des Universe-Typsystem vorbereitet. Es waren die entprechenden Integerkonstanten in escjava.ast.TagConstants definiert und die Universe-Modifier hätten dadurch beim Parsing ignoriert werden sollen. Die Implementierung war aber fehlerhaft, so dass es zum Absturz kam, wenn man Universe-Modifier benutzte.

Da ich das Universe-Typsystem aber innerhalb von Javafe und nicht in Escjava einbauen wollte (siehe Ort der Implementierung 3.1.1), entfernte ich diesen Code wieder.

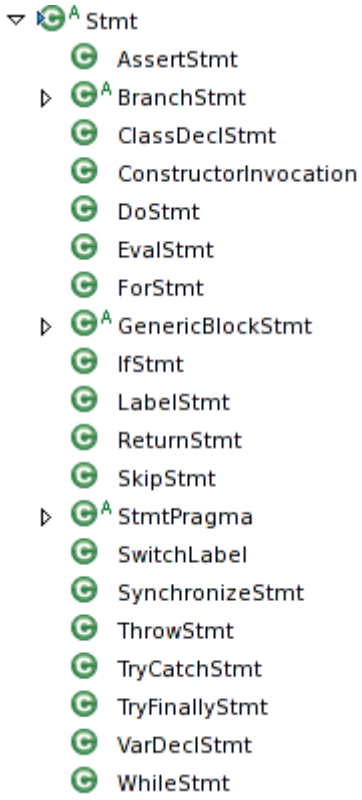

Abbildung 2.4: Abstract-Syntax-Tree für Anweisungen

Abbildung 2.5: Abstract-Syntax-Tree für Ausdrücke

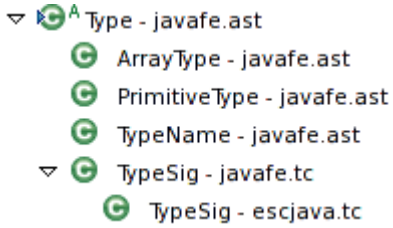

Abbildung 2.6: Klassenhierarchie der TypSig-Klassen

Kapitel 3

Implementierung des Universe-Typsystem in ESC/Java 2

3.1 Design-Entscheidungen

3.1.1 Ort der Implementierung

Eine grundsätzliche Frage ist, ob die Behandlung der Universe-Typ-Modifier im Javafrontend Javafe oder im JML-Backend Escjava platziert werden soll. Bisher wurde fast nichts von der JML-Verarbeitung in Javafe gemacht. Die Universe-Modifier können aber an Orten stehen, wo Modifierpragmas (dazu gehören die Universe-Modifier, siehe 2.4) nicht erlaubt sind. Deshalb muss der Parser in Javafe sowieso angepasst werden. Zudem wäre es gut, wenn die Universe-Modifier auch als Schlüsselwörter direkt im Javacode benutzt werden können. Dies erfordert ebenfalls Änderungen in Javafe. Sollte man dann nicht auch das Prüfen der Universe-Modifier in Javafe machen?

Man könnte das Parsen in Javafe machen und das Prüfen in Escjava, was aber nicht sehr effizient wäre, weil man beinahe die selbe Kontrolllogik wie im Typchecker von Javafe noch einmal braucht und viele Methoden überschreiben müsste. In den überschriebenen Methoden müsste man Code duplizieren, weil es an vielen verschiedenen Stellen kleine Änderungen braucht und man deshalb nicht einfach die Super-Implementation aufrufen kann. Es ist daher einfacher und effizienter, die Checks direkt in Javafe einzubauen. Zudem können dadurch auch andere Anwendungen, welche Javafe als Javafrontend brauchen, die Universe-Modifier benutzen, ohne diese noch prüfen zu müssen. Deshalb wurde das Prüfen der Universe-Modifier ebenfalls in Javafe eingebaut.

3.1.2 Art der Repräsentation

Die zweite wichtige Entscheidung ist die Art der Repräsentation der Universe-Typ-Modifier. Man kann für jede Klassendefinition einen peer-, einen rep- und einen readonly-Typ (genauer je eine TypeSig) erzeugen, was das eigentliche Typchecking vereinfacht. Die Frage ist, wie dieses Vorgehen die erweiterten statischen Checks beeinflusst. Die andere Möglichkeit ist die Universe-Typ-Modifier an die entsprechenden AST-Knoten zu hängen, so wie es mit den anderen Modifiern auch getan wird. Dieser Ansatz ist sicherer und einfacher einzubauen, weil man die schrittweise Vervollständigung der Typen nicht anpassen muss und die Auswirkungen auf das statische Checking klar absehbar sind. Dafür muss das eigentliche Prüfen der Universe-Modifier an vielen Stellen getan werden, was beim anderen Ansatz zentral in wenigen Methoden wie isSubtypOf der Klasse TypeSig getan werden könnte.

Nachdem das Parsing implementiert war, wurden beide Ansätze getestet. Wie erwartet erwies sich der Ansatz mit den 3 Typen pro Klassendefinition als kompliziert. Für jede Zustandsänderung

einer der 3 TypeSigs wurden alle 3 angepasst (mehr Details in 4.2). Es traten Probleme mit vordefinierten Typen wie java.lang.String auf, deren Ursache nicht gefunden wurde (vielleicht wurde eine Zustandsänderung der TypSigs übersehen). Die befürchteten Einflüsse auf das statische Prüfen konnten deshalb nicht verifiziert werden.

Der zweite Ansatz, die Universe-Modifier an die AST-Knoten zu hängen, wurde vor allem wegen seinen vorhersagbaren Auswirkungen gewählt.

3.2 Kommandozeilenparameter

Die Option '-universes' aktiviert die Unterstützung des Universe-Typsystem in ESC/Java 2. Danach kann angegeben werden, wo die Universe-Modifier stehen können. Es gibt die Möglichkeiten 'comment' (in JML-Spezifikationen), 'keyword' (als Schlüsselwort im Code) und 'commentandkeyword' (beides). Danach kann 'readonlyforpurector' stehen, was bewirkt, dass der Standard für die Parameter von puren Konstruktoren readonly ist. Dies ist nützlich, weil die mitgelieferten JML-Spezifikationen für die Standard-Javaklassen davon ausgehen, obwohl dies falsch ist. Dahinter kann noch 'nochecks' stehen, was bewirkt, dass die Anwendung der Universe-Modifier nicht auf Korrektheit geprüft wird. Dabei entspricht '-universes' der Kombination '-universes comment'.

Zusätzlich kann mit der Option '-specs file.jar' angegeben werden, welche Spezifikationsdatei benutzt wird. Da die standardmässige Datei jmlspecs.jar noch nicht ganz kompatibel ist mit ESC/Java 2, sollte man '-specs escspecs.jar' benutzen (natürlich mit Pfadangabe).

3.3 Details der gewählten Implementation

Beim Parsen werden die Universe-Modifier direkt an die AST-Knoten gehängt. Diese Knoten sind CastExpr, InstanceOfExpr, NewArrayExpr, NewInstanceExpr, MethodeDecl (für die Universe-Modifier des Rückgabewerts) oder GenericVarDecl (mit Subtypen FieldDecl, FormalParaDecl und LocalVarDecl).

Die Universe-Modifier werden durch Integerwerte kodiert, diese sind PEER, REP, READONLY und IMPL_PEER, welche in der Klasse javafe.parser.TagConstants definiert sind. In der Klasse javafe.tc.FlowInsensitiveChecks findet das Typchecking statt, dabei werden für das Prüfen der Universe-Typen die Regeln aus Kapitel 1 angewandt. Die für Ausdrücke ermittelten Universe-Modifier werden ebenfalls im Universe-Feld des entsprechenden Knoten gespeichert. Um das Prüfen zu vereinfachen, können auch NULLLIT und THISEXPR für die beiden entsprechenden AST-Knoten im Universe-Feld gespeichert werden.

Alle Stellen im Code, die an der Verarbeitung der Universe-Typ-Modifier beteiligt sind, sind mit dem String '//dw' markiert und können so schnell gefunden werden.

3.3.1 javafe.Options:

Diese Klasse analysiert die Kommandozeilenparameter und setzt dabei die bool'sche Variable useUniverseTypeSystem und den Integerwert universeLevel. Der Integerwert ist durch 2 teilbar, wenn die Universe-Modifier in den JML-Kommentaren erkannt werden sollen. Er ist durch 3 teilbar, wenn sie als Schlüsselwörter direkt im Javacode vorkommen können. Der Wert ist durch 5 teilbar, wenn der Parameter 'nochecks' gesetzt ist, welcher bewirkt, dass die Universe-Modifier geparst, aber nicht auf ihre korrekte Anwendung geprüft werden. All dies geschieht in der Methode processOption.

Eine Instanz dieser Klasse befindet sich im statischen Feld options der Klasse Tool, so dass die Kommandozeilenparameter von überall her im Programm zugreifbar sind.

3.3.2 javafe.parser.TagConstants:

Hier werden die Integer Konstanten definiert, welche den Typ der AST-Knoten kodieren. Neben REP, PEER und READONLY wurde noch die Konstante IMPL_PEER definiert, als Standard

für nicht explizit angegebene Universe-Typ-Modifier. Alle Schlüsselwörter müssen im Stringarray keywordStrings aufgezählt werden (in Definitionsreihenfolge).

```
public static final int IMPL_PEER = TYPEMODIFIERPRAGMA + 1;
/* .... */
public static final int REP = VOLATILE + 1;
public static final int PEER = REP + 1;
public static final int READONLY = PEER + 1;
```

3.3.3 javafe.parser.ParseUtil:

In dieser Klasse befindet sich die Methode parseModifiers, welche auch die Universe-Modifier behandelt. Diese (höchstens 2) werden als Seiteneffekt im Feld universeArray gespeichert. Das mit //(1) markierte if-Statement behandelt Universe-Modifier in Kommentaren, welche vom Pragmaparser als Modifierpragma mit entsprechend gesetztem auxVal Feld zurückgegeben werden. Das mit //(2) markierte if-Statement kümmert sich um die Universe-Typ-Modifier, welche als Schlüsselwort im Code vorkommen.

```
public int[] universeArray = {0,0};

public int parseModifiers(/*@ non_null @*/ Lex l) {
    boolean seenPragma = false;
    int nof_universeModifiers=0;
    universeArray[0]=0;
    universeArray[1]=0;

    int modifiers = Modifiers.NONE;

    getModifierLoop:
    for(;;) {
        if (l.ttype == TagConstants.MODIFIERPRAGMA) {

            //(1)   handle universe modifiers returned from PragmaParser
            if (universeLevel%2==0 && l.auxVal instanceof ModifierPragma) {
                int tag = ((ModifierPragma) l.auxVal).getTag();
                if (tag==TagConstants.PEER || tag==TagConstants.REP ||
                        tag==TagConstants.READONLY) {
                    if (nof_universeModifiers>1) {
                        if (universeLevel%5!=0)
                            ErrorSet.error(((ModifierPragma) l.auxVal).getStartLoc(),
                                    "too many Universe type modifiers");
                        l.getNextToken();
                        continue getModifierLoop;
                    }
                    universeArray[nof_universeModifiers++]=tag;
                    l.getNextToken();
                    continue getModifierLoop;
                }
            }
            if (! seenPragma) {
                seqModifierPragma.push();
            seenPragma = true;
            }
```

```
            seqModifierPragma.addElement(l.auxVal);
            l.getNextToken();
            continue getModifierLoop;
        } else

            //(2)   handle universe modifiers that are used as keywords
            if (universeLevel%3==0 && (l.ttype==TagConstants.PEER ||
                    l.ttype==TagConstants.REP || l.ttype==TagConstants.READONLY)) {
                if (nof_universeModifiers>1) {
                    if (universeLevel%5!=0)
                        ErrorSet.error(l.startingLoc,"too many Universe type modifiers");
                    l.getNextToken();
                    continue getModifierLoop;
                }
                universeArray[nof_universeModifiers++]=l.ttype;
                l.getNextToken();
                continue getModifierLoop;
            } else {
                int i = getJavaModifier(l,modifiers);
                if (i != 0) {
                    modifiers |= i;
                    continue getModifierLoop;
                }
            }
        // Next token is not a modifier
        if (! seenPragma)
            modifierPragmas = null;
        else
            modifierPragmas
                = ModifierPragmaVec.popFromStackVector(seqModifierPragma);
        return modifiers;
    }
}
```

Die Methode parseUniverses ruft parseModifiers auf und gibt für alle Modifier ausser den Universe-Typ-Modifiern eine Fehlermeldung aus. Dies ist nützlich für Stellen, an denen nur Universe-Modifier erlaubt sind (Cast-, Instanceof- und New-Statement).

```
public void parseUniverses(Lex l) {
    int loc = l.startingLoc;
    int jmods = parseModifiers(l);
    if (jmods!=0)
        ErrorSet.error(loc,"no java modifiers allowed");
    if (modifierPragmas!=null && modifierPragmas.size()!=0)
        ErrorSet.error(loc,"only Universe type modifiers allowed");
}
```

Die AST-Knoten können um neue Felder erweitert werden, indem man ein statisches Feld vom Typen ASTDecoration definiert. Die ASTDecoration-Instanz speichert dazu den aktuellen Wert eines statischen Zählers und inkrementiert danach den Zähler. Der Wert des statischen Zählers entspricht dabei der Anzahl solcher ASTDecorationen. Der gespeicherte Wert dient als Index in ein Object-Array, welches ein Feld der AST-Knoten ist. ASTDecoration bietet eine set- und eine

get-Methode an, welche das Object-Array des übergebenen AST-Knoten an der Stelle des Indexes schreibt, bzw. liest.

Dieser Mechanismus wird in den Methoden setUniverse bzw. getUniverse benutzt, um die Universe-Modifier an die AST-Knoten zu hängen. Für Arrays muss zusätzlich noch der Universe-Typ der Elemente gespeichert werden, dazu wird ebenfalls eine ASTDecoration benutzt, um die Methoden set- bzw. getElementUniverse zur Verfügung zu stellen.

```
private static ASTDecoration universeDecoration
    = new ASTDecoration("universeDecoration");

public static ASTNode setUniverse(ASTNode i, int u) {
    universeDecoration.set(i,new Integer(u));
    return i;
}

public static int getUniverse(ASTNode i) {
    Object o = universeDecoration.get(i);
    if (o instanceof Integer)
        return ((Integer) o).intValue();
    return 0;
}
```

Wenn die Universe-Modifier geparst wurden und sich im Feld universeArray befinden, wird die folgende Variante von setUniverse mit dem Array als Argument aufgerufen. Sie gibt Fehlermeldungen aus, falls die Modifier nicht zum mitgegebenen Typen des ASTNode passen. Fehlt der Modifier (Wert im Array ist 0), dann wird IMPL_PEER benutzt.

```
public static ASTNode setUniverse(ASTNode i, int[] a, Type t, int loc) {
    int tag = t.getTag();
    boolean reportErrors = Tool.options!=null && Tool.options.universeLevel%5!=0;
    if (a!=null && !(tag==TagConstants.ARRAYTYPE)) {
        if (t instanceof PrimitiveType) {
            if (a[0]!=0 && reportErrors)
                ErrorSet.error(loc,"primitive types must not have Universe modifiers");
            return i;
        }
        if (a[0]!=0)
            universeDecoration.set(i,new Integer(a[0]));
        else
            universeDecoration.set(i,new Integer(TagConstants.IMPL_PEER));
        if (a[1]!=0 && reportErrors)
            ErrorSet.error(loc,"only array types can have 2 Universe modifiers");
    }
    else if (a!=null && tag==TagConstants.ARRAYTYPE) {
        ArrayType at = (ArrayType ) t;
        //primitive types need only the array modifier
        if (at.elemType instanceof PrimitiveType) {
            if (a[0]==0) {
                universeDecoration.set(i,new Integer(TagConstants.IMPL_PEER));
                return i;
            }
            else if (a[1]!=0 && reportErrors)
                ErrorSet.error(loc,"only one Universe modifier allowed for arrays "
                        +"of primitve type");
```

```
            universeDecoration.set(i,new Integer(a[0]));
        }
        else {
            if (a[0]==0) {
                universeDecoration.set(i,new Integer(TagConstants.IMPL_PEER));
                elementUniverseDecoration.set(i,new Integer(TagConstants.IMPL_PEER));
                return i;
            }
            else if(a[1]==0) {
                if (a[0]==TagConstants.REP) {
                    if (reportErrors)
                        ErrorSet.error(loc,"rep not allowed for array elements");
                    a[0]=TagConstants.IMPL_PEER;
                }
                universeDecoration.set(i,new Integer(TagConstants.IMPL_PEER));
                elementUniverseDecoration.set(i,new Integer(a[0]));
            } else {
                if (a[1]==TagConstants.REP) {
                    if (reportErrors)
                        ErrorSet.error(loc,"rep not allowed for array elements");
                    a[1]=TagConstants.IMPL_PEER;
                }
                universeDecoration.set(i,new Integer(a[0]));
                elementUniverseDecoration.set(i,new Integer(a[1]));
            }
        }
    }
    if (reportErrors  && getUniverse(i)==TagConstants.READONLY &&
            (i.getTag()==TagConstants.NEWINSTANCEEXPR
                    || i.getTag()==TagConstants.NEWARRAYEXPR))
        ErrorSet.error(i.getStartLoc(),"readonly not allowed for new expression, except"
                +"for array element modifier");
    return i;
}
```

3.3.4 javafe.parser.ParseExpr:

In der Methode parseExpression wird das Instanceof-Statement behandelt. Dabei werden zuerst die gefundenen Universe-Modifier zwischengespeichert und am Ende an den neuen AST-Knoten angehängt. Das Feld universeArray wird nach Aufrufen von parseUniverses bzw. parseModifier immer geklont, da bis zum Erstellen des entsprechenden AST-Knoten weitere parse-Methoden aufgerufen werden, welche universeArray ändern könnten.

```
if( op == TagConstants.INSTANCEOF ) {
    l.getNextToken();

    // parse the universe modifiers
    int[] localUniverseArray=null;
    if (useUniverses) {
        parseUniverses(l);
        localUniverseArray = (int[]) this.universeArray.clone();
    }
    Type t = parseType( l );
```

```
    e = InstanceOfExpr.make( e, t, locOp );

     // set the universe modifiers
     if (useUniverses)
         setUniverse(e, localUniverseArray, t, locOp);

    // Now go to check following op
    continue getOp;
}
```

Analog zum eben gezeigten Instanceof-Statement wird auch in parseNewExpression und in parseCastExpression zuerst parseUniverses aufgerufen, dann universeArray geklont und am Ende die Universe-Modifier aus dem geklonten Array per setUniverse an den neuen AST-Knoten gehängt.

3.3.5 javafe.parser.ParseStmt:

In addStmt muss an einer Stelle das Feld universeArray auf {0,0} gesetzt werden, weil entlang des entsprechenden Kontrollpfades nie parseModifiers aufgerufen wird und deshalb noch alte Werte im Array stehen können. Natürlich wird wieder nach jedem parseModifiers das Feld universeArray geklont. Zudem muss ein If-Statement, welches testet ob Modifiers angetroffen wurden, nun auch Universe-Modifier umfassen.

```
if (modifiers != Modifiers.NONE || pmodifiers != null || universeArray[0]!=0) ;
```

In parseCatches muss sichergestellt werden, dass der Universe-Modifier readonly ist.

```
if (useUniverses && getUniverse(arg)==TagConstants.IMPL_PEER)
    setUniverse(arg,TagConstants.READONLY);
else if (useUniverses && getUniverse(arg)!=TagConstants.READONLY) {
    setUniverse(arg,TagConstants.READONLY);
    ErrorSet.error(arg.getStartLoc(),"only readonly allowed for catch clauses");
}
```

Etwas speziell verhält sich die Methode addVarDeclStmt. Sie parst Variablendeklarationen ohne den am Anfang stehenden Typen, welcher ein Parameter der Methode ist. Die so erkannten Variablendefinitionen werden als Seiteneffekt im Vektor seqStmt gespeichert. Diese Methode erwartet, dass die Universe-Modifier bei ihrem Aufruf bereits in universeArray gespeichert sind, sie ruft selbst nie parseModifiers auf.

Zu Beginn klont sie universeArray und hängt dann an jeden Variablendeklarationsknoten die Universe-Modifier aus dem geklonten Array.

Die Methode parseFormalParaDecl verhält sich wieder wie gewohnt. Nach jedem Aufruf von parseModifiers wird universeArray geklont und später mit dem geklonten Array die Universe-Modifier für den neu erstellten formalen Parameter gesetzt.

3.3.6 javafe.parser.Parse:

Die beiden Methoden parseTypeDeclElemIntoSeqTDE und parseFormalParameterList verhalten sich auch wie gewohnt. Nach jedem Aufruf von parseModifiers wird universeArray geklont und später mit dem geklonten Array die Universe-Modifier für den neu erstellten AST-Knoten gesetzt.

3.3.7 javafe.tc.FlowInsensiveChecks:

Diese Klasse ist für das Typchecking zuständig. Zuerst werden ein paar Felder definiert, um sich Kontextinformationen zur aktuell behandelten Methode zu merken. Zudem wird das Feld useUniverses im Konstruktor initialisiert.

```
protected boolean useUniverses=false;
boolean inPure=false;
boolean inCtor=false;
boolean inStatic=false;
int universeReturnType = 0;
int universeElementReturnType = 0;
```

Die Methode checkUniverseAssignability gibt Fehlermeldungen aus, wenn der rechte ASTNode auf Grund der Universe-Modifier nicht dem linken zugewiesen werden kann. Es wird dazu getestet, ob die rechte Seite ein Subtyp der linken Seite ist. Für Cast- und Instanceof-Statement wird in checkUniverseCastability getestet, ob eine der Seiten ein Subtyp der anderen ist. Beide Methoden benutzen dazu die Methode universeIsSubtypeOf.

```
public static void checkUniverseAssignability(ASTNode l, ASTNode r) { /* .... */ }

public static void checkUniverseCastability(ASTNode l, ASTNode r) { /* .... */ }

public static boolean universeIsSubtypeOf(int sub,int sup){
    if (sub==TagConstants.NULLLIT || sub==0 || sup==0)
        return true;
    if (sub==TagConstants.THISEXPR || sub==TagConstants.IMPL_PEER)
        sub=TagConstants.PEER;
    if (sup==TagConstants.THISEXPR || sup==TagConstants.IMPL_PEER)
        sup=TagConstants.PEER;
    return (sup==sub || sup==TagConstants.READONLY);
}
```

Die folgende Methode bestimmt, ob eine Methode oder ein Konstruktor pur ist. Da dem Javafrontend Javafe der pure-Modifier nicht bekannt ist, muss sie in escjava.tc.FlowInsensitiveChecks überschrieben werden,

```
protected boolean isPure(RoutineDecl rd) {
    return false;
}
```

Die folgende Methode ist die Implementation des Universe-Typ-Kombinator. Man benutzt aber meistens die überladene Variante, welche zwei AST-Knoten als Argument nimmt und dann diese Variante aufruft.

```
protected static int universeTypeCombiner(int l, int r) {
    switch (l) {
    case TagConstants.THISEXPR: //dw: special case for l=peer=this
    {
        return r;
    }
    case TagConstants.PEER:
    case TagConstants.IMPL_PEER:
    {
```

```
        if (r==TagConstants.PEER || r==TagConstants.IMPL_PEER)
            return TagConstants.PEER;
        return TagConstants.READONLY;
    }
    case TagConstants.REP:
    {
        if (r==TagConstants.PEER || r==TagConstants.IMPL_PEER)
            return TagConstants.REP;
        return TagConstants.READONLY;
    }
    case TagConstants.READONLY:
    {
        return TagConstants.READONLY;
    }
    }

    return 0; //dummy
}
```

In einem statischen Kontext darf nie der Universe-Modifier rep benutzt werden, was durch aufrufen der Methode checkNoRepInStaticContext sichergestellt wird.

```
protected void checkNoRepInStaticContext(ASTNode n) {
    if (inStatic && javafe.parser.ParseUtil.getUniverse(n)==TagConstants.REP)
        ErrorSet.error(n.getStartLoc(),"usage of rep not allowed in static contexts");
}
```

Die Methode checkUniverseForField wird für jede Felddeklaration aufgerufen und stellt sicher, dass kein statisches Feld rep ist. Zudem gibt sie eine Warnung aus, dass implizit peer als Standard benutzt wird, obwohl es readonly sein sollte.

```
boolean impl_peerInStaticCautionThrown = false;

protected void checkUniverseForField(GenericVarDecl gvd) {
    int mod=javafe.parser.ParseUtil.getUniverse(gvd);
    if (mod!=0 && Modifiers.isStatic(gvd.modifiers) && mod!=TagConstants.READONLY) {
        if (mod==TagConstants.REP)
            ErrorSet.error(gvd.getStartLoc(),"static fields cannot be of universe type rep");
        if (!impl_peerInStaticCautionThrown && mod==TagConstants.IMPL_PEER) {
            ErrorSet.caution(gvd.getStartLoc(),"using 'implicit peer' as default for static"
                    +"fields, but should be readonly");
            impl_peerInStaticCautionThrown=true;
        }
    }
}
```

Die folgende Methode kopiert die Universe-Modifier von source nach dest. Dies wird z.B. benutzt um die Modifier der linken Seite einer Zuweisung an den Zuweisungsknoten zu hängen oder die Modifier eines Ausdrucks in Klammern an den Klammerknoten. Für den Refinement-Prozess müssen ebenfalls Modifier kopiert werden (siehe 3.3.11).

```
public static void copyUniverses(ASTNode dest, ASTNode source) { /* .... */ }
```

Die nächste Methode bestimmt die kleinste obere Schranke zweier Universe-Modifier. Das ist der erste gemeinsame Supertyp der beiden. Dies ist nützlich für das bestimmen der Universe-Typen bei bedingter Zuweisung wie '(b) ? (peer T) x: (rep T) x' oder bei Arrayinitialisierern.

```
public static int leastUpperUniverseBound(int l, int r) {
    if (l==TagConstants.THISEXPR || l==TagConstants.IMPL_PEER)
        l=TagConstants.PEER;
    if (r==TagConstants.THISEXPR || r==TagConstants.IMPL_PEER)
        r=TagConstants.PEER;

    if (l==r)
        return r;
    else if (l==TagConstants.NULLLIT)
        return r;
    else if (r==TagConstants.NULLLIT)
        return l;
    else
        return TagConstants.READONLY;
}
```

Für einen Ausdruck wie this.x.y wird ein FieldAccess-Knoten erstellt. Dieser Knoten repräsentiert einen Zugriff auf das Feld y, dabei ist das Target des Zugriffs im Feld od gespeichert. Im Feld od des FieldAccess y ist in diesem Beispiel ein FieldAccess x gespeichert, dessen od-Feld enthält einen This-Ausdruck.

Die Methode determineUniverseForFieldAccess bestimmt mit Hilfe des Typ-Kombinator den Universe-Typen eines Feldzugriffs. Der AST-Knoten MethodInvocation hat ebenfalls dieses od-Feld, welches das Target bestimmt. Die Methode determineUniverseForMethodInvocation bestimmt den Universe-Typ des Rückgabewerts und stellt sicher, dass nur pure Methoden auf readonly-Targets aufgerufen werden und dass rep-Argumente nur auf dem Target this benutzt werden. Da es nur einen erlaubten Fall[1] gibt, bei dem die Universe-Typen der formalen Parameter mit dem Target kombiniert einen anderen Universe-Typ ergeben, ist dieser Fall direkt eingebaut, anstatt den Universe-Typ-Kombinator auf jedes Argument anzuwenden.

```
public static void determineUniverseForFieldAccess(FieldAccess fa) {
    //if target has no universe modifier, set it to impl_peer
    if (fa.od!=null && javafe.parser.ParseUtil.getUniverse(fa.od)==0)
        javafe.parser.ParseUtil.setUniverse(fa.od,TagConstants.IMPL_PEER);
    //if decl has no universe modifier, set to default
    if (!(fa.decl.type instanceof PrimitiveType) &&
            javafe.parser.ParseUtil.getUniverse(fa.decl)==0)
        javafe.parser.ParseUtil.setUniverse(fa.decl,
                new int[] {0,0},fa.decl.type,Location.NULL);
    //use universeTypeCombiner
    if (javafe.parser.ParseUtil.getUniverse(fa.decl)!=0) {
        javafe.parser.ParseUtil.setUniverse(fa,universeTypeCombiner(fa.od,fa.decl));
        if (javafe.parser.ParseUtil.getElementUniverse(fa.decl)!=0)
            javafe.parser.ParseUtil.setElementUniverse(fa,
                    javafe.parser.ParseUtil.getElementUniverse(fa.decl));
    }
}
```

[1] Dieser Fall ist ein rep-Target mit formalem Parameter vom Typ peer. Der aktuelle Parameter muss den Typ rep haben (rep*peer=rep).

```
public void determineUniverseForMethodInvocation(MethodInvocation mi) {
    //if target has no universe modifier, set it to impl_peer
    if (mi.od!=null && javafe.parser.ParseUtil.getUniverse(mi.od)==0)
        javafe.parser.ParseUtil.setUniverse(mi.od,TagConstants.IMPL_PEER);
    int od = javafe.parser.ParseUtil.getUniverse(mi.od);
    //if decl has no universe modifier, set to default
    if (!(mi.decl.returnType instanceof PrimitiveType) &&
            javafe.parser.ParseUtil.getUniverse(mi.decl)==0)
        javafe.parser.ParseUtil.setUniverse(mi.decl,new int[] {0,0},
                mi.decl.returnType,Location.NULL);
    //use universeTypeCombiner
    if (javafe.parser.ParseUtil.getUniverse(mi.decl)!=0) {
        javafe.parser.ParseUtil.setUniverse(mi,universeTypeCombiner(mi.od,mi.decl));
        if (javafe.parser.ParseUtil.getElementUniverse(mi.decl)!=0)
            javafe.parser.ParseUtil.setElementUniverse(mi,
                    javafe.parser.ParseUtil.getElementUniverse(mi.decl));

    }
    if (od==TagConstants.READONLY && !isPure(mi.decl))
        ErrorSet.error(mi.getStartLoc(),"only pure methods can be called on readonly targets");
    boolean repIsError = od!=TagConstants.THISEXPR;
    for (int i = 0; i<mi.args.size(); i++) {
        if (!(mi.decl.args.elementAt(i).type instanceof PrimitiveType) &&
            javafe.parser.ParseUtil.getUniverse(mi.decl.args.elementAt(i))==0)
            javafe.parser.ParseUtil.setUniverse(mi.decl.args.elementAt(i),new int[] {0,0},
                mi.decl.args.elementAt(i).type,Location.NULL);
        int decl_arg = javafe.parser.ParseUtil.getUniverse(mi.decl.args.elementAt(i));
        //if the target is rep and a method is defined as foo(peer T x), then
        //the actual argument has to be rep T. (universeTypeCombiner: rep*peer=rep)
        if (od==TagConstants.REP &&
                (decl_arg==TagConstants.PEER || decl_arg==TagConstants.IMPL_PEER)) {
            if (javafe.parser.ParseUtil.getUniverse(mi.args.elementAt(i))!=TagConstants.REP)
                ErrorSet.error(mi.args.elementAt(i).getStartLoc(),
                        "the argument has to be rep: rep*peer=rep");
        }
        else {
            checkUniverseAssignability(mi.decl.args.elementAt(i),mi.args.elementAt(i));
            if (repIsError && decl_arg==TagConstants.REP) {
                ErrorSet.error(mi.getStartLoc(),
                        "methods with rep arguments can only be called on target this");
                repIsError=false;
            }
        }
    }
}
```

Für Zugriffe auf Arrayelemente wird ebenfalls der Universe-Typ-Kombinator angewandt. Der Ausdruck im Feld array des ArrayRefExpr-Knoten besitzt den Universe-Typen des Ausdrucks ohne Elementzugriff, der nun mit dem Arrayelement-Modifier kombiniert werden muss. Dies wird in der Methode determineUniverseForArrayRefExpr gemacht und das Resultat an den übergebenen ArrayRefExpr-Knoten gehängt.

```
public static void determineUniverseForArrayRefExpr(ArrayRefExpr r) {
    javafe.parser.ParseUtil.setUniverse(r,universeTypeCombiner(
```

```
            javafe.parser.ParseUtil.getUniverse(r.array),
    javaafe.parser.ParseUtil.getElementUniverse(r.array)
    ));
}
```

Bei Feldzuweisungen wird geprüft, dass das Target nicht readonly ist und dass das Feld nicht rep ist (ausser das Target ist this). Dies geschieht in der Methode checkBinaryExpr (siehe folgenden Code). Das gleiche muss noch für ArrayRefExpr-Knoten getan werden. Für VariableAccess-Knoten ist dies nicht nötig, da sie immer schreibbar sind.

```
if (javafe.parser.ParseUtil.getUniverse(fa.od)==TagConstants.READONLY)
    ErrorSet.error(fa.getStartLoc(),"cannot assign to field of readonly target");
else if (javafe.parser.ParseUtil.getUniverse(fa.decl)==TagConstants.REP
        && javafe.parser.ParseUtil.getUniverse(fa.od)!=TagConstants.THISEXPR)
    ErrorSet.error(fa.getStartLoc(),"cannot assign to rep field of target different from this");
```

3.3.8 escjava.ast.TagConstants:

Es werden neue Integerkonstanten für die alternativen Varianten \peer, \rep und \readonly definiert, welche nur in JML-Kommentaren vorkommen können.

```
public static final int PEER2 = ALSO_REQUIRES + 1;
public static final int READONLY2 = PEER2 + 1;
public static final int REP2 = READONLY2 + 1;
```

Das Array esckeywords muss ebenfalls um diese neuen Schlüsselwörter erweitert werden.

3.3.9 escjava.parser.EscPragmaParser:

Der Pragmaparser parst die JML-Spezifikationen. Werden in der Methode getNextPragmaHelper Universe-Modifier (in normaler oder alternativer Form) angetroffen, dann werden sie in einen Modifierpragma gepackt. Da das Prüfen in Javafe gemacht wird, müssen die alternativen Schlüsselwörter durch die normalen ersetzt werden.

```
case TagConstants.PEER:
case TagConstants.READONLY:
case TagConstants.REP:
    dst.ttype = TagConstants.MODIFIERPRAGMA;
    dst.auxVal = SimpleModifierPragma.make(tag, loc);
    scanner.getNextToken();
    break;
case TagConstants.PEER2:
    dst.ttype = TagConstants.MODIFIERPRAGMA;
    dst.auxVal = SimpleModifierPragma.make(TagConstants.PEER, loc);
    break;
case TagConstants.READONLY2:
    dst.ttype = TagConstants.MODIFIERPRAGMA;
    dst.auxVal = SimpleModifierPragma.make(TagConstants.READONLY, loc);
    break;
case TagConstants.REP2:
    dst.ttype = TagConstants.MODIFIERPRAGMA;
    dst.auxVal = SimpleModifierPragma.make(TagConstants.REP, loc);
    break;
```

Zudem werden in der Methode parsePrimaryExpression die Universe-Modifier für die JML-Funktion \type(T) an den entsprechenden AST-Knoten gehängt. Dies wird in JML-Spezifikationen wie `//@ requires x <: \type(peer T)` benutzt. Diese Information wird aber in der von Simplify getriebenen Logik noch nicht ausgewertet. Es würde also noch nicht erkannt, wenn die vorher deklarierte Vorbedingung verletzt wird.

3.3.10 escjava.tc.FlowInsensitiveChecks:

Die Methode isPure wird überschrieben, weil die benötigte Information in Javafe nicht zur Verfügung steht.

```
protected boolean isPure(RoutineDecl rd) {
    return Utils.isPure(rd);
}
```

An gewissen Stellen müssen determineUniverseForFieldAccess, determineUniverseForMethodInvocation und determineUniverseForArrayRefExpr aufgerufen werden, weil für wenige Spezialfälle die Superklassenimplementationen nicht aufgerufen wird.

3.3.11 escjava.RefinementSequence:

Die JML-Spezifikationen (oder Teile davon) können auch in externen Dateien stehen, in so genannten Refinements. Die Refinements werden automatisch eingelesen, falls sie sich auf dem Sourcepfad befinden, einen Klassennamen als Präfix und einen der folgenden Suffixe haben: .refines-java, .refines-spec, .refines-jml, .spec, .jml, .java-refined, .spec-refined und .jml-refined. Die Refinements können wiederum durch weitere Refinements erweitert werden. So entstehen ganze Refinement-Sequenzen, welche kombiniert werden, um die resultierende Typdeklaration zu erhalten. Diese Refinements dürfen sich gegenseitig nicht widersprechen.

Intern werden auch die im Javacode enthaltenen JML-Spezifikationen in ein erstes Refinement transformiert. Aus dem Javacode wird eine reine Java-Klassendeklaration erstellt, welche danach mit diesem ersten Refinement kombiniert wird. Die so erhaltene Deklaration wird dann mit den weiteren Refinements kombiniert.

Die Klasse escjava.RefinementSequence bietet die beiden Methoden combineFields und combineRoutine an, welche ihrerseits die Methode combineUniverses aufrufen um die Universe-Typen der Refinements zu kombinieren.

```
void combineFields(FieldDecl newfd, FieldDecl fd) {
    // combine universe of refined newfd with universe of fd
    if (useUniverses)
        combineUniverses(newfd,fd);
    /* .... */
}

//combine the universe modifiers of the old node with them of the
//refined new node, into the old node!!!
private void combineUniverses(ASTNode newNode, ASTNode oldNode) {
    int n=javafe.parser.ParseUtil.getUniverse(newNode);
    int o=javafe.parser.ParseUtil.getUniverse(oldNode);
    if (n==0)
        return;
    if (o==0 || o==TagConstants.IMPL_PEER)
        javafe.parser.ParseUtil.setUniverse(oldNode,n);
    else if(o!=n) {
        ErrorSet.error(newNode.getStartLoc(),"cannot refine to "+TagConstants.toString(n)
```

```
            +", already defined as "+TagConstants.toString(o));
        return;
    }
    //now the same for the element type (if an array)
    int n2=javafe.parser.ParseUtil.getElementUniverse(newNode);
    int o2=javafe.parser.ParseUtil.getElementUniverse(oldNode);
    if (n2==0)
        return;
    if (o2==0 || o2==TagConstants.IMPL_PEER)
        javafe.parser.ParseUtil.setElementUniverse(oldNode,n2);
    else if(o2!=n2)
        ErrorSet.error(newNode.getStartLoc(),"cannot refine to "+TagConstants.toString(n)+" "
            +TagConstants.toString(n2)+", already defined as "+TagConstants.toString(o)
        +" "+TagConstants.toString(o2));
}
```

Kapitel 4

Fazit

4.1 Erfahrungen

Für mich war dieses Projekt sehr lehrreich, da ich noch nie mit derart umfangreichem Sourcecode konfrontiert war. Das Programm ist so umfangreich, dass ich mich darauf beschränken musste, nur die für mich relevanten Teile genauer zu betrachten. Ich brauchte Monate, um das Programm einigermassen zu verstehen, da es keine vernünftige Dokumentation gibt. Das Dokument [3], welches die Architektur erklären würde, ist leider immer noch leer. Dass sowohl die lexikographische Analyse, wie auch das Parsing von Hand implementiert wurden, und nicht mit einem entsprechenden Generator, machte die ganze Sache noch unüberschaubarer. Weil das Parsing der Universe-Modifier unabhängig vom Überprüfen dieser ist, war ich froh, als ich endlich mit dem Programmieren beginnen konnte, nachdem ich das Parsing einigermassen verstanden hatte. Danach testete ich zwei Arten für die Repräsentation der Universe-Modifier. Aus Softwareengineer-Sicht ist der gewählte Ansatz nicht ganz optimal (siehe 4.2), aber aus Zeitgründen war er die einzige Möglichkeit, um eine funktionierende Implementation erstellen zu können. Danach machte ich mich schliesslich ans Typchecking. Es gibt noch offene Punkte (siehe 4.3), aber da ich schon wesentlich länger daran gearbeitet habe, als geplant, musste ich einen Schlussstrich ziehen.

Es war für mich sehr schwer, den Zeitaufwand für einzelne Arbeiten abzuschätzen. Denn am meisten Zeit verbrauchte ich jeweils, um den Code verstehen zu können und zu testen, ob mein Ansatz funktionieren könnte. Die dafür nötige Zeit konnte ich nur schlecht abschätzen und das jeweilige Implementieren war danach vergleichsweise schnell erledigt.

4.2 Kritik am gewählten Ansatz

Der gewählte Ansatz, die Universe-Typen direkt an die AST-Knoten zu hängen, ist aus Softwareengineering-Sicht nicht optimal. Er hat den Vorteil, dass er recht unabhängig von der restlichen Logik ist. Ich wusste also, dass dieser Ansatz die anderen Teile des Programms nicht beeinflusst. Diese Unabhängigkeit von der restlichen Logik ist aber zugleich auch der Nachteil dieser Lösung. Da an vielen Stellen, wo Typen getestet werden, nur die TypeSigs übergeben werden, mussten jeweils die Unverse-Typen noch separat getestet werden. Dieser Ansatzt machte also viele kleine Änderungen am Code nötig.

Der bessere Ansatz wäre, aus jeder Typdeklaration drei TypeSigs zu erstellen, einen peer-Typ, einen rep-Typ und einen readonly-Typ. Dies erfordert nur eine minimale Änderung im Parsing. Wie in 2.7 beschrieben, sind die vielen Zustandsänderungen der TypSigs das Problem. Für jede Zustandsänderung einer TypSig wurden alle wichtigen Felder dieser TypSig auf die anderen beiden TypSigs kopiert. Ich habe dabei noch eine vierte TypSig benutzt, welche implizite peer-Typen darstellt. Jede TypeSig hat ein Feld key, welches auf den entsprechenden impliziten peer-Typ zeigt, der implizite peer-Typ zeigt dabei auf sich selbst. Ich erstellte für jede erzeugte TypeSig einen Eintrag in einer Hash-Tabelle mit dem impliziten peer-Typen als Schlüssel und dem Tripel peer-,

rep- und readonly-TypSig als Wert. So konnten nach jeder Änderung in einer der vier die restlichen angepasst werden, indem man die Hash-Tabelle befragt. Dieser Ansatz sollte funktionieren, ich brachte ihn aber nicht zum laufen. Es gab für mich nicht erklärbare Fehlermeldungen bezüglich extern definierter Klassen wie java.lang.String. Das Auflösen der Typen ist sehr komplex, so kann es gut sein, dass ich eine Zustandsänderung der TypSigs übersehen habe.

Der Vorteil dieses Ansatzes wäre gewesen, dass man an den entscheidenden Stellen wo, die TypSigs für das Typchecking verglichen werden, auch die Information über die Universe-Typen zur Verfügung hätte. So könnte man das Typchecking für die Universe-Typen mit einigen zentralen Veränderungen implementieren. Das ist aus Softwareengineering-Sicht schöner als bei der gewählten Lösung.

Der Nachteil wäre aber ein enormer Zeitaufwand gewesen, um sicher zu stellen, dass die restliche Programmlogik nicht davon beeinflusst wird. Diese Änderung könnte Einflüsse auf die Beweiserlogik haben. Und mich mit der auch noch zu beschäftigen, lag in der gegebenen Zeit nicht drin. Da ich schon in der Testimplementation auf Probleme stiess, gab ich diesen Ansatz auf.

4.3 Offene Punkte

Ich habe es nicht mehr geschafft, Universe-Modifier vor statischen Methodenaufrufen zu unterstützen. Eine statische Methode sollte mit peer T.m() oder rep T.m() aufgerufen werden können. Ich habe begonnen, das Parsing daran anzupassen, die entsprechenden Stellen sind mit dem String '//dw code for static calls' markiert. Der entsprechende Code ist auskommentiert, da er noch nicht perfekt ist. Alle statischen Methodenaufrufe werden als peer T.m() interpretiert und es darf kein Universe-Modifier davor stehen.

Die Universe-Typen werden von Simplify beim erweiterten statischen Checking noch nicht benutzt. Dies war auch nicht Inhalt dieser Arbeit. Deshalb wird nicht erkannt, wenn eine Vorbedingung wie `//@ requires x <: \type(peer T)` verletzt wird.

4.4 Testfälle

Ich habe ein paar Testfälle erstellt. Sie prüfen die unter 1.6 aufgestellten Regeln.

Das folgende File Test1.java testet viele dieser Fälle. Die dabei ausgegebenen Fehlermeldungen sind danach aufgeführt.

```
public class Test1 {
        rep Test1 repTest1;
        String  hello;
        Object oo1;
        peer Object oo2;
        rep String err1 = new peer String();
        rep Test1 err2 = new Test1();
        Test1 err3 = new rep Test1();
        readonly Test1 hhh = new Test1();
        Test1 hhhh = new peer Test1();
        readonly Test1 roh1;
        /*@ \rep @*/ Test1 rh1;
        /*@ rep @*/ Test1 rh2;
        rep Object o1;
        readonly Object o2 = new peer Object();
        rep Object err4 = new peer Object();
        peer Test1 ph1;

        void foo() {
```

```
        o1=new rep Object();
        peer Object o3 = new peer Object();
        Object o4;
        String  ipstr ;
        readonly Test1 h1 = new peer Test1();
        peer readonly Test1[] prh = (peer readonly Test1[]) new peer peer Test1[3];
        rep Test1 rh2;
        int i = rh2.hhhh.hhhh.hashCode();
        System.out.println( ipstr );
        oo2 = rh2.o1; //err5
        oo1 = rh2.o1; //err6
        o3 = new rep Object(); //err7
        o4 = new rep Object(); // err8
        Object todo = rh2.o1; //err9
        rep String err10 = (rep String) ipstr ;
        o1 = prh[0]; //err11
        prh = new rep peer Test1[6]; //err12

        rh2 instanceof rep Test1;
        //  peer String ps;
        readonly readonly Test1[] rrha;
        rh2 = new rep Test1();
        rrha = new peer peer Test1[4];
        roh1 = (readonly Test1) ph1;
        rrha = (readonly readonly Test1[]) rrha;
        boolean b1 = roh1 instanceof peer Test1;
        boolean b2 = rrha instanceof peer peer Test1[];
}

static void main(){
        rep Object err13;
        readonly Object xxx;
        xxx = new rep Object(); //err14
        xxx = (rep Object) new Object(); //err15&16
        xxx instanceof rep Object; //err17
}
static rep Object err18_19(rep String s,readonly Test1 h) {
        h.foo (); //err20
        return new rep Object(); //err21
}

static Object caution1;
static Object no_caution;
static readonly Object no_err2;
static rep Object err22;
Test1(rep Object o) { //err23
        readonly Object err24 = new readonly Object();
}

Test1() {
        new Test1(new Object()); //err25
        peer Object err26 = this.foo1(new Object(),new Object(),new Object());
}
```

```
        peer Object foo1 (peer Object o1, rep Object o2, readonly Object o3) {
                return new rep Object(); //err27
                rep Object err28 = (new peer Object()).toString();
        }

        /*@ pure rep @*/ Object foo2(Object o1, peer Object err29, rep String err30) {
                try {
                        1/0;
                } catch (java.lang.Exception e) {
                        e.toString(); //err31
                }

                rep Object o2 = null;
                Object o3 = null;

                //tests for writability of fields
                hhh.hhh=hhhh; //err32 (ro*ro)
                hhh.hhhh=hhhh; //err33 (ro*peer)
                hhhh.hhh=hhhh; //should work (peer*readonly)
                hhhh.o1=hhhh; //err34 (peer*rep)
                hhhh.hhhh.hhhh.hhhh.hhh=hhhh; //should work
                repTest1.repTest1=repTest1; //err35 (rep*rep)
                repTest1.hhhh=hhhh; //err36 (rep*peer)=rep
                repTest1.hhhh=hhh; //err37 (rep=ro)
                return new rep Object(); //not_found_err38 nicht purer ctor
        }

        //dw test rep target for method with peer param:
        void repTest(peer Object p) {
                rep Test1 r = new rep Test1();
                r.repTest(p); //err38 param has to be rep*peer=rep
                r.repTest(r); //this one works
        }
}
```

Fehlermeldungen für Test1.java (Test1.out.txt):

```
ESC/Java version ESCJava-CURRENT-CVS
/home/emule/eclipseprojects/test/src/Test1.java:67:
Error: readonly not allowed for new expression, except for array element modifier
                readonly Object err24 = new readonly Object();
                                        ^

    [0.07 s 4506616 bytes]

Test1 ...
/home/emule/eclipseprojects/test/src/Test1.java:7:
Error: cannot assign peer to rep
        rep String err1 = new peer String();
                          ^

/home/emule/eclipseprojects/test/src/Test1.java:8:
Error: cannot assign [peer] to rep
        rep Test1 err2 = new Test1();
                         ^

/home/emule/eclipseprojects/test/src/Test1.java:9:
```

```
Error: cannot assign rep to [peer]
        Test1 err3 = new rep Test1();
                     ^

/home/emule/eclipseprojects/test/src/Test1.java:17:
Error: cannot assign peer to rep
        rep Object err4 = new peer Object();
                          ^

/home/emule/eclipseprojects/test/src/Test1.java:30:
Error: cannot assign readonly to peer
                oo2 = rh2.o1; //err5
                      ^

/home/emule/eclipseprojects/test/src/Test1.java:31:
Error: cannot assign readonly to [peer]
                oo1 = rh2.o1; //err6
                      ^

/home/emule/eclipseprojects/test/src/Test1.java:32:
Error: cannot assign rep to peer
                o3 = new rep Object(); //err7
                     ^

/home/emule/eclipseprojects/test/src/Test1.java:33:
Error: cannot assign rep to [peer]
                o4 = new rep Object(); // err8
                     ^

/home/emule/eclipseprojects/test/src/Test1.java:34:
Error: cannot assign readonly to [peer]
                Object todo = rh2.o1; //err9
                              ^

/home/emule/eclipseprojects/test/src/Test1.java:35:
Error: A [peer] instance can never be of universe type rep
                rep String err10 = (rep String) ipstr;
                                                ^

/home/emule/eclipseprojects/test/src/Test1.java:36:
Error: cannot assign readonly to rep
                o1 = prh[0]; //err11
                     ^

/home/emule/eclipseprojects/test/src/Test1.java:37:
Error: cannot assign rep to peer
                prh = new rep peer Test1[6]; //err12
                      ^

/home/emule/eclipseprojects/test/src/Test1.java:51:
Error: usage of rep not allowed in static contexts
                rep Object err13;
                    ^

/home/emule/eclipseprojects/test/src/Test1.java:53:
Error: usage of rep not allowed in static contexts
                xxx = new rep Object(); //err14
                      ^

/home/emule/eclipseprojects/test/src/Test1.java:54:
Error: A [peer] instance can never be of universe type rep
                xxx = (rep Object) new Object(); //err15&16
                                   ^

/home/emule/eclipseprojects/test/src/Test1.java:54:
Error: usage of rep not allowed in static contexts
                xxx = (rep Object) new Object(); //err15&16
```

```
                    ^
/home/emule/eclipseprojects/test/src/Test1.java:55:
Error: usage of rep not allowed in static contexts
                xxx instanceof rep Object; //err17
                ^
/home/emule/eclipseprojects/test/src/Test1.java:57:
Error: usage of rep not allowed in static contexts
        static rep Object err18_19(rep String s,readonly Test1 h) {
        ^
/home/emule/eclipseprojects/test/src/Test1.java:57:
Error: usage of rep not allowed in static contexts
        static rep Object err18_19(rep String s,readonly Test1 h) {
                                   ^
/home/emule/eclipseprojects/test/src/Test1.java:58:
Error: only pure methods can be called on readonly targets
                h.foo(); //err20
                ^
/home/emule/eclipseprojects/test/src/Test1.java:59:
Error: usage of rep not allowed in static contexts
                return new rep Object(); //err21
                       ^
/home/emule/eclipseprojects/test/src/Test1.java:62:
Caution: using 'implicit peer' as default for static fields, but should be readonly
        static Object caution1;
               ^
/home/emule/eclipseprojects/test/src/Test1.java:65:
Error: static fields cannot be of universe type rep
        static rep Object err22;
                   ^
/home/emule/eclipseprojects/test/src/Test1.java:66:
Error: rep not allowed in constructor signature
        Test1(rep Object o) { //err23
              ^
/home/emule/eclipseprojects/test/src/Test1.java:71:
Error: cannot assign [peer] to rep
                new Test1(new Object()); //err25
                          ^
/home/emule/eclipseprojects/test/src/Test1.java:72:
Error: cannot assign [peer] to rep
                peer Object err26 = this.foo1(new Object(),new Object( ...
                                              ^
/home/emule/eclipseprojects/test/src/Test1.java:76:
Error: This routine must return peer, not rep
                return new rep Object(); //err27
                ^
/home/emule/eclipseprojects/test/src/Test1.java:77:
Error: cannot assign peer to rep
                rep Object err28 = (new peer Object()).toString();
                                   ^
/home/emule/eclipseprojects/test/src/Test1.java:80:
Error: arguments of pure methods must have readonly universe type
        /*@ pure rep @*/ Object foo2(Object o1, peer Object err29, rep ...
                                                 ^
/home/emule/eclipseprojects/test/src/Test1.java:80:
```

```
Error: arguments of pure methods must have readonly universe type
  ... (Object o1, peer Object err29, rep String err30) {
                                                ^

/home/emule/eclipseprojects/test/src/Test1.java:84:
Error: only pure methods can be called on readonly targets
                        e.toString(); //err31
                        ^

/home/emule/eclipseprojects/test/src/Test1.java:91:
Error: cannot assign to field of readonly target
                hhh.hhh=hhhh; //err32 (ro*ro)
                ^

/home/emule/eclipseprojects/test/src/Test1.java:92:
Error: cannot assign to field of readonly target
                hhh.hhhh=hhhh; //err33 (ro*peer)
                ^

/home/emule/eclipseprojects/test/src/Test1.java:94:
Error: cannot assign to rep field of target different from this
                hhhh.o1=hhhh; //err34 (peer*rep)
                ^

/home/emule/eclipseprojects/test/src/Test1.java:96:
Error: cannot assign to rep field of target different from this
                repTest1.repTest1=repTest1; //err35 (rep*rep)
                ^

/home/emule/eclipseprojects/test/src/Test1.java:97:
Error: cannot assign [peer] to rep
                repTest1.hhhh=hhhh; //err36 (rep*peer)=>rep=peer
                              ^

/home/emule/eclipseprojects/test/src/Test1.java:98:
Error: cannot assign readonly to rep
                repTest1.hhhh=hhh; //err37 (rep=ro)
                              ^

/home/emule/eclipseprojects/test/src/Test1.java:105:
Error: the argument has to be rep: rep*peer=rep
                r.repTest(p); //err38 param has to be rep*peer=rep
                          ^

Caution: Turning off extended static checking due to type error(s)
  [0.34 s 4961800 bytes total] (aborted)
2 cautions
38 errors
```

Das File ArrayTest.java testet die Arrayinitialisierer.

```
// tests for array initializers, 8 errors,
// last 3 because of rep in static context
public class ArrayTest {
        int[] a = {1,2,3};
        peer peer ArrayTest[] pph = {new peer ArrayTest(), new rep ArrayTest()};
        peer peer ArrayTest[] pph2 = new peer peer ArrayTest[]
                {new peer ArrayTest(), new rep ArrayTest()};
        peer readonly ArrayTest[] pphx = {new peer ArrayTest(), new rep ArrayTest()};
        peer readonly ArrayTest[] pph2x = new peer readonly ArrayTest[]
                {new peer ArrayTest(), new rep ArrayTest()};

        void foo(){
```

```
        peer peer ArrayTest[] pph = {new peer ArrayTest(), new rep ArrayTest()};
        peer peer ArrayTest[] pph2 = new peer peer ArrayTest[]
                {new peer ArrayTest(), new rep ArrayTest()};
        new peer peer ArrayTest[] {new peer ArrayTest(), new rep ArrayTest()};
        peer readonly ArrayTest[] pphx = {new peer ArrayTest(), new rep ArrayTest()};
        peer readonly ArrayTest[] pph2x = new peer readonly ArrayTest[]
                {new peer ArrayTest(), new rep ArrayTest()};
        new peer readonly ArrayTest[] {new peer ArrayTest(), new rep ArrayTest()};
    }

    static void main() {
        peer readonly ArrayTest[] pph = {new peer ArrayTest(), new rep ArrayTest()};
        peer readonly ArrayTest[] pph2 = new peer readonly ArrayTest[]
                {new peer ArrayTest(), new rep ArrayTest()};
        new peer readonly ArrayTest[] {new peer ArrayTest(), new rep ArrayTest()};
    }
}
```

ArrayTest.out.txt enthält die Fehlermeldungen zu ArrayTest.java.

```
ESC/Java version ESCJava-CURRENT-CVS
    [0.051 s 4461792 bytes]

ArrayTest ...
/home/emule/eclipseprojects/test/src/ArrayTest.java:4:
Error: cannot assign peer readonly to peer peer
        peer peer ArrayTest[] pph = {new peer ArrayTest(), new rep Arr ...
                                    ^

/home/emule/eclipseprojects/test/src/ArrayTest.java:5:
Error: cannot assign peer readonly to peer peer
        peer peer ArrayTest[] pph2 = new peer peer ArrayTest[] {new pe ...
                                                ^

/home/emule/eclipseprojects/test/src/ArrayTest.java:10:
Error: cannot assign peer readonly to peer peer
                peer peer ArrayTest[] pph = {new peer ArrayTest(), new ...
                                            ^

/home/emule/eclipseprojects/test/src/ArrayTest.java:11:
Error: cannot assign peer readonly to peer peer
  ... ] pph2 = new peer peer ArrayTest[] {new peer ArrayTest(), new rep ...
                                        ^

/home/emule/eclipseprojects/test/src/ArrayTest.java:12:
Error: cannot assign peer readonly to peer peer
                new peer peer ArrayTest[] {new peer ArrayTest(), new r ...
                                         ^

/home/emule/eclipseprojects/test/src/ArrayTest.java:19:
Error: usage of rep not allowed in static contexts
  ... est [] pph = {new peer ArrayTest(), new rep ArrayTest()};
                                         ^

/home/emule/eclipseprojects/test/src/ArrayTest.java:20:
Error: usage of rep not allowed in static contexts
  ... ArrayTest[] {new peer ArrayTest(), new rep ArrayTest()};
                                        ^

/home/emule/eclipseprojects/test/src/ArrayTest.java:21:
Error: usage of rep not allowed in static contexts
```

```
  ... ArrayTest[] {new peer ArrayTest(), new rep ArrayTest()};
                                         ^

Caution: Turning off extended static checking due to type error(s)
  [0.076 s 4494376 bytes total] (aborted)
1 caution
8 errors
```

Das File ArrayElementTest.java testet die Benutzung von Arrayelementen.

```
// ==> 6 Fehler !!!!
// bei readonly als array modifier können keine elemente
// zugewiesen werden, nur ganze arrays.
class ArrayElementTest {
        peer peer ArrayElementTest[] pp = new peer peer ArrayElementTest[5];
        rep peer ArrayElementTest[] rp = new rep peer ArrayElementTest[5];
        readonly peer ArrayElementTest[] rop = new rep peer ArrayElementTest[5];
        readonly readonly ArrayElementTest[] roro;

        void foo(peer ArrayElementTest a) {
                a.rp[0].pp[0]=this; //err1
                peer ArrayElementTest p = a.pp[0].rp[0]; //err2
                rep ArrayElementTest r = a.rp[0].pp[0]; //err3
                a.pp[0].pp[0].pp[0] = a.pp[0].pp[0]; //sollte gehen
                a.roro = a.rop; //sollte gehen
                a.rop[0] = a; //err4
                this.rop[0] = a; //err5
                a.rp = new rep peer ArrayElementTest[6]; //err6
                this.rp = new rep peer ArrayElementTest[6]; //sollte gehen
                this.rp[0] = new rep ArrayElementTest(); //sollte gehen
        }
}
```

Die dabei erzeugten Fehlermeldungen sind im File ArrayElementTest.out.txt.

```
ESC/Java version ESCJava-CURRENT-CVS
    [0.045 s 4448440 bytes]

ArrayElementTest ...
/home/emule/eclipseprojects/test/src/ArrayElementTest.java:11:
Error: cannot assign to elements of a readonly array reference
                a.rp[0].pp[0]=this; //err1
                ^

/home/emule/eclipseprojects/test/src/ArrayElementTest.java:12:
Error: cannot assign readonly to peer
                peer ArrayElementTest p = a.pp[0].rp[0]; //err2
                                          ^

/home/emule/eclipseprojects/test/src/ArrayElementTest.java:13:
Error: cannot assign readonly to rep
                rep ArrayElementTest r = a.rp[0].pp[0]; //err3
                                         ^

/home/emule/eclipseprojects/test/src/ArrayElementTest.java:16:
Error: cannot assign to elements of a readonly array reference
                a.rop[0] = a; //err4
```

```
                ^
/home/emule/eclipseprojects/test/src/ArrayElementTest.java:17:
Error: cannot assign to elements of a readonly array reference
                this.rop[0] = a; //err5
                ^
/home/emule/eclipseprojects/test/src/ArrayElementTest.java:18:
Error: cannot assign to rep field of target different from this
                a.rp = new rep peer ArrayElementTest[6]; //err6
                ^
Caution: Turning off extended static checking due to type error(s)
  [0.071 s 4460304 bytes total] (aborted)
1 caution
6 errors
```

Um die Refinement-Sequenzen (siehe 3.3.11) zu testen, wird JMLTest.java benutzt.

```
public class JMLTest {
        public Object o;
        public peer Object err1_with_spec;

        public static void main(String[] args) {
                JMLTest j = new JMLTest();
                j.foo(new String[] {"foo","bar"});
        }

        public void foo(String[] args) {
                o = new rep Object(); //err1_without_spec
                o = new Object(); //err2_with_spec
                o = args[0]; //err3_with_spec
        }

        public peer Object err4(rep Object err5) {}
}
```

Zusätzlich wird noch ein Refinement benutzt. Das folgende Refinement ist im File JMLTest.refines-java gespeichert.

```
//@ refine "JMLTest.java";

//dw second part of refinement sequence test
public class JMLTest {
        public /*@ rep @*/ Object o;
        public rep Object err1_with_spec;
        public void foo( /*@ peer peer @*/ String[] args);
        public rep Object err4(peer Object err5);
}
```

Die Ausgabe dieses Refinement-Tests befindet sich im File JMLTest.out.txt.

```
ESC/Java version ESCJava-CURRENT-CVS
/home/emule/eclipseprojects/test/src/JMLTest.refines-java:6:
Error: cannot refine to rep, already defined as peer
        public rep Object err1_with_spec;
```

```
                    ^
/home/emule/eclipseprojects/test/src/JMLTest.refines-java:8:
Error: cannot refine to rep, already defined as peer
        public rep Object err4(peer Object err5);
               ^
/home/emule/eclipseprojects/test/src/JMLTest.refines-java:8:
Error: cannot refine to peer, already defined as rep
        public rep Object err4(peer Object err5);
                                       ^
    [0.05 s 4469424 bytes]

JMLTest ...
/home/emule/eclipseprojects/test/src/JMLTest.java:13:
Error: cannot assign [peer] to rep
                o = new Object(); //err2_with_spec
                    ^
/home/emule/eclipseprojects/test/src/JMLTest.java:14:
Error: cannot assign peer to rep
                o = args[0]; //err3_with_spec
                    ^
Caution: Turning off extended static checking due to type error(s)
  [0.119 s 4381424 bytes total] (aborted)
1 caution
5 errors
```

Das letzte Testfile JMLTest2.java prüft die Anwendung von Universe-Modifiern innerhalb von JML-Spezifikationen. Das Testprogramm erzeugt keinen Fehler.

```
//dw test jml-operator \typeof(), the precond. violation isn't detected!
public class JMLTest2 {

        /*@ requires \typeof(t) <: \type(rep JMLTest2); @*/
        public void foo(readonly JMLTest2 t) {
        }

        public static void main() {
                peer JMLTest2 jp = new JMLTest2();
                jp.foo(jp); //error, precondition violated
        }
}
```

Literaturverzeichnis

[1] W. Dietl, P. Müller und D. Schregenberger. *Universe Type System – Quick-Reference.* Software Component Technology Group, ETH Zürich, 2005.
http://www.sct.inf.ethz.ch/research/universes/tools/juts-quickref.pdf

[2] P. Müller und A. Poetzsch-Heffter. *Universes: A type system for alias and dependency control.* Technical Report 279, Fernuniversität Hagen, 2001.

[3] J. Kiniry. *Extending ESC/Java 2.* (noch nicht abgeschlossen). University College Dublin.
http://secure.ucd.ie/products/opensource/ESCJava2/ESCTools/docs/Escjava2-Extending/Escjava2-Extending.pdf

[4] J. Kiniry. *Using ESC/Java 2 - Architecture, Hints, and Tricks.* University College Dublin.
http://secure.ucd.ie/documents/slides/Using_ESCJava2.pdf

[5] Dokumentationen zu ESC/Java 2.
http://secure.ucd.ie/products/opensource/ESCJava2/docs.html