Technische Universität Berlin
Fakultät IV - Elektrotechnik und Informatik
Institut für Softwaretechnik und Theoretische Informatik

Formalisierung und Analyse eines Typsystems für ObjectTeams/Java mit dem Beweiswerkzeug Isabelle

Diplomarbeit von

Rebekka Oeters

Prinzregentenstraße 12
10717 Berlin
Matrikelnummer: 188531
lytosska@cs.tu-berlin.de

Erstgutachter: Prof. Dr. Stefan Jähnichen
Zweitgutachter: Dr. Florian Kammüller

Die selbständige und eigenhändige Anfertigung versichere ich an Eides Statt.

Berlin, den 19.09.2003
Unterschrift:

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

In dieser Arbeit wird ein Typsystem für ObjectTeams/Java - die Einbettung des Programmiermodells Object Teams in die Programmiersprache Java - modelliert. Dieses Typsystem wird hinsichtlich seiner Typzuverlässigkeit analysiert. Zwei zentrale Konzepte von Object Teams sind Teamklassen - Teams - und Rollenklassen. Es wird betrachtet, ob das Teams-Konstrukt eine inhärente Grenze hinsichtlich der Referenzierbarkeit von Rollenobjekten darstellt.

Die Modellierung eines Typsytems für ObjectTeams/Java basiert auf der Formalisierung einer abstrakten Syntax für ObjectTeams/Java. Sie besteht aus der Definition eines Modells instanzbasierter Typen, der Beschreibung von Beziehungen zwischen diesen und Regeln zur Typisierung von ObjectTeams/Java-Konstrukten. Die statische Analyse von ObjectTeams/Java wird durch die Formalisierung von Wohlgeformtheitsbedingungen und den Nachweis einer statischen Schichtenaussage über die Wohlkonstruiertheit von Typen einschließlich der Bildung von Rollentypen durch Rollenklassen einer Teamklasse - einer Teamschicht - abgeschlossen. Basierend auf diesen Formalisierungen der statischen Aspekte von ObjectTeams/Java und der Modellierung einer operationalen Semantik für ObjectTeams/Java wird eine Aussage über die Typzuverlässigkeit des Typsytems formuliert und analysiert. Ein weiterer Gesichtspunkt hinsichtlich der Analyse des Typsystems ist die Formulierung einer Aussage über die Kontrolle von Referenzen auf Rollenobjekte, die das Teams-Konstrukt ausübt.

Die Formalisierungen und Analysen werden unter Verwendung des interaktiven Beweiswerkzeugs Isabelle durchgeführt, und zwar mit Isabelle/HOL - der Einbettung einer Logik höherer Ordnung in das generische Beweiswerkzeug Isabelle. Alle Definitionen der Modellierung sind in Isabelle/HOL kodiert. Der größte Teil der formulierten Aussagen über die Modellierung ist nachgewiesen.

Eine Motivation für die Modellierung eines Typsystems für ObjectTeams/Java ist die formale Analyse, ob statisch sichergestellt werden kann, dass die kovariante Redefinition von Rollenklassen bei der Bildung von Subteamklassen nicht zu Typfehlern führt. Die kovariante Redefinition von Rollenklassen im Kontext einer Subteamklasse hat zur Folge, dass Methodenparameter vom Typ einer Rollenklasse kovariant redefiniert werden. Die kovariante Redefinition von Methoden ist im allgemeinen nicht typsicher, wie zum Beispiel die Analyse des Typsystems der Programmiersprache Eiffel gezeigt hat [Szy98]. Durch die Modellierung eines Typsystems für ObjectTeams/Java soll formal gezeigt werden, dass das Sprachdesign von ObjectTeams/Java in Kombination mit einer Menge von Wohlgeformtheitsbedingungen das Auftreten von Typfehlern trotz der kovarianten Redefinition von Rollenklassen verhindert. Wenn dies bewiesen werden kann, dann ist die Typsicherheit von ObjectTeams/Java nachgewiesen. Die Typsicherheit von ObjectTeams/Java ist im Rahmen dieser Arbeit weitgehend nachgewiesen.

Eine weitere Motivation für die Modellierung eines Typsystems für ObjectTeams/Java ist die damit mögliche Spezifikation dieser Wohlgeformtheitsbedingungen. Es werden Einschränkungen hinsichtlich der Wohlgeformtheit von ObjectTeams/Java-Programmen definiert, denen ObjectTeams/Java-Programme genügen müssen, damit zur Laufzeit dieser Programme keine unerwarteten Typfehler auftreten. Bei diesen Wohlgeformtheitsbedingungen handelt es sich um Überprüfungen, die beim Kompilieren von Programmen durchgeführt werden.

Weiterhin ermöglicht die Modellierung eines Typsystems für ObjectTeams/Java eine formale Analyse der Eigenschaft von Object Teams, Referenzen auf in Teamobjekten enthaltene Rollenobjekte zu kontrollieren. Unter der Voraussetzung, dass die Typsicherheit von ObjectTeams/Java komplett gezeigt werden kann, wird nachgewiesen, dass alle Rollenobjekte nur innerhalb der sie umschließenden Teaminstanz referenziert werden. Dies wird als *Confinement* von Rollenobjekten bezeichnet. Die Voraussetzung beim Nachweis der Aussage über das *Confinement* von Rollenobjekten muss getroffen werden, weil die Typsicherheit von ObjectTeams/Java im Rahmen dieser Arbeit nicht vollständig nachgewiesen ist. Diese Eigenschaft von Object Teams macht seine Verwendung im Kontext von Security-Anwendungen interessant, in denen der Zugriff auf bestimmte in Objekten gekapselte Informationen kontrolliert werden soll.

Der Nachweis der Typsicherheit der Programmiersprache ObjectTeams/Java garantiert noch keine typsichere Ausführung von ObjectTeams/Java-Programmen, weil der von einem ObjectTeams/Java-Compiler produzierte Bytecode fehlerhaft oder verfälscht sein könnte. Die Untersuchung der Typsicherheit von ObjectTeams/Java kann jedoch als Teil der Sprachanalyse betrachtet werden, weil dabei überprüft wird, ob das Design der Sprache zusammen mit einer Menge von Wohlgeformtheitsbedingungen das Auftreten von statisch überprüfbaren Typfehlern ausschließt.

Die Modellierung und Analyse eines Typsystems für ObjectTeams/Java ist an die in [KNO+02a] und [KNO+02b] vorgestellten Formalisierungen der Programmiersprache Java angelehnt. Im Rahmen dieser

Arbeit sind folgende Bestandteile der Programmiersprache ObjectTeams/Java formalisiert.

- Team- und Rollenklassen
- Vererbung zwischen Teamklassen
- Zweifacherben von Rollenklassen
- Ein Modell instanzbasierter Typen, wobei nur Rollentypen betrachtet werden, die abhängig von der in Team- und Rollenklassendeklarationen enthaltenen Referenz auf das aktuelle beziehungsweise umschließende Teamobjekt gebildet werden
- Basisklassen mit statischen Methoden
- Abstrakte Klassen und Methoden

Im Gegensatz zu [KNO+02a] und [KNO+02b] sind im Rahmen dieser Arbeit folgende Sprachbestandteile von Java nicht in der Formalisierung der Programmiersprache ObjectTeams/Java enthalten.

- Interfaces und Arrays
- Pakete
- Überladen von Methoden
- Sichtbarkeitsmodifikatoren auf Klassen- und Methodenebene
- Detaillierte Behandlung von Exceptions durch Formalisierung der `try`- und `catch`-Anweisungen und benutzerdefinierte Exception-Klassen
- Statische Attribute von Klassen
- Numerische und logische Operatoren, alternative Schleifenanweisungen, komplexe Blockstrukturen und Behandlung mehrerer Rückgabeanweisungen innerhalb von Methodenkörpern
- *Definite Assignment* [GJS+00] (§16)
- *Abrupt Termination* [GJS+00] (§14.1)

Die Arbeit ist wie folgt aufgebaut. In Kapitel 2 wird das Programmiermodell Object Teams vorgestellt. Die Konsequenzen der kovarianten Redefinition von Rollenklassen bei der Bildung von Subteamklassen für die Modellierung eines Typsystems für ObjectTeams/Java werden herausgearbeitet. In Kapitel 3 werden Motivationen für die Typisierung und Typüberprüfung von Programmen im Rahmen der Entwicklung von Software dargestellt. Es wird eine Vorgehensweise bei der Formalisierung eines Typsystems für eine Programmiersprache mit dem Ziel, dessen Typzuverlässigkeit nachzuweisen, erläutert. In Kapitel 4 wird das Beweiswerkzeug Isabelle und die innerhalb dieser Arbeit verwendete Instanziierung von Isabelle mit einer Logik höherer Ordnung namens Isabelle/HOL vorgestellt. Die Formalisierungen von Java in Isabelle/HOL, an die diese Arbeit angelehnt ist, werden hinsichtlich der Vorgehensweise bei der Einbettung einer Programmiersprache in eine Logik vorgestellt. Die Einordnung und der grobe Aufbau der Einbettung von ObjectTeams/Java in Isabelle/HOL werden dargestellt. Diese Einbettung wird in den Kapiteln 5 bis 9 vorgestellt. In Kapitel 5 wird eine abstrakte Syntax für ObjectTeams/Java formalisiert. Die Konzepte der Team- und Rollenklasse und ihre Vererbungsbeziehungen werden definiert. In Kapitel 6 wird ein Typsystem für ObjectTeams/Java modelliert. Ein Modell instanzbasierter Typen und die Beziehungen zwischen diesen werden vorgestellt. Die Regeln zur Typisierung von ObjectTeams/Java-Konstrukten werden formalisiert. In Kapitel 7 ist eine statische Analyse von ObjectTeams/Java enthalten. Es werden Wohlgeformtheitsbedingungen für ObjectTeams/Java-Programme definiert. Desweiteren wird eine statische Schichtenaussage bezüglich der Wohlkonstruiertheit von Typen einschließlich der Bildung von Rollentypen durch Rollenklassen einer Teamschicht nachgewiesen. In Kapitel 8 wird eine Semantik für ObjectTeams/Java formalisiert. In Kapitel 9 werden die Typzuverlässigkeit des modellierten Typsystems für ObjectTeams/Java und das *Confinement* von Rollenobjekten analysiert. In Kapitel 10 ist eine Bewertung der formulierten Aussagen, eine Einschätzung hinsichtlich des Aufwandes der noch nachzuweisenden Aussagen und ein Ausblick bezüglich möglicher weiterführender Arbeiten enthalten.

2 Das Programmiermodell Object Teams

Ein Überblick über das Programmiermodell Object Teams wird in Abschnitt 2.1 gegeben. Die Sicht auf Object Teams als Rahmenwerk stellt den Fokus dieser Arbeit dar. Sie wird an einem Beispiel erläutert. Die für die Arbeit zentralen Konzepte der Team- und Rollenklasse werden in den Abschnitten 2.2 und 2.3 vorgestellt.

2.1 Überblick über das Programmiermodell Object Teams

Object Teams [Her03a] stellen ein neues Programmiermodell zur Kapselung einer Menge von kollaborierenden Klassen dar. Die Module zur Gruppierung dieser kollaborierenden Klassen werden als Teamklassen bezeichnet, die miteinander agierenden Klassen innerhalb von Teamklassen als Rollenklassen. Object Teams ermöglichen die Wiederverwendung von in Teamklassen gekapseltem Verhalten zum einen durch die Vererbung von Eigenschaften von Teamklassen bei der Bildung von Subteamklassen. Zum anderen wird in Teamklassen definiertes Verhalten durch die Integration von Teamklassen in Anwendungen mittels deklarativer Bindungen von Rollenklassen an Klassen dieser sogenannten Basisanwendungen wiederverwendet. Dabei werden Object Teams zur Kapselung von über strukturelle Entitäten einer Anwendung verteilten Gesichtspunkten (*crosscutting concerns*) verwendet mit dem Ziel, Anwendungen modularer zu gestalten. Die in Object Teams gekapselten Aspekte von Anwendungen können in anderen Kontexten wiederverwendet werden. Durch eine Integration von Object Teams-Anwendungen können bestehende Basisanwendungen a-posteriori nicht invasiv adaptiert werden.

Der Fokus dieser Arbeit liegt auf der Betrachtung von Teamklassen als Rahmenwerke, deren Verhaltensabläufe durch die innerhalb einer Teamklasse enthaltenen, miteinander agierenden Rollenklassen und durch Verhaltensdefinitionen auf der Ebene von Teamklassen definiert sind.

Ein Ausschnitt aus der Modellierung eines Rahmenwerkes für Brettspiele mit Object Teams ist in Abbildung 1 enthalten. Die Implementierung dieses Ausschnittes ist im Anhang zu finden. Die vollständige Implementierung des Brettspiel-Rahmenwerkes und zwei Instanziierungen zur Implementierung der Brettspiele „TicTacToe“ und „Vier Gewinnt“ sind unter `http://www.objectteams.org` erhältlich.

Die Teamklasse `BoardGame` definiert ein allgemeines, teilweise noch abstrakt gehaltenes Verhalten und die strukturellen Bestandteile, die Brettspielen gemein sind. Auf Teamklassenebene sind in diesem Ausschnitt nur die Methoden `initialize` zur Initialisierung eines Brettspiels, die Methode `checkConsistencyWithRules` zur Überprüfung, ob ein Spielzug regelkonform ist, und die noch abstrakt gehaltene Methode `addRules` zum Hinzufügen von Regelobjekten zu einem Brettspiel enthalten. Letztere stellt einen *hot spot* des Brettspiel-Rahmenwerkes dar, der bei einer Instanziierung des Rahmenwerkes anwendungsspezifisch implementiert werden muss.

Die strukturellen Bestandteile eines Brettspiels sind durch die in der Teamklasse `BoardGame` enthaltenen Rollenklassen modelliert. Die Rollenklasse `Board` dient zur Modellierung eines Brettes eines Brettspiels. Die Rollenklassen `Player` und `Token` modellieren Spieler beziehungsweise von Spielern während eines Spiels zu setzende Spielsteine. Die Rollenklasse `Rule` dient zur Modellierung von Regeln, die während eines Spiels überprüft werden sollen.

Das Brettspiel-Rahmenwerk wird durch Bildung der Subteamklasse `TicTacToe` instanziiert. Diese erbt alle Bestandteile der Teamklasse `BoardGame`, also sowohl Attribute und Methoden, als auch alle in der Superteamklasse enthaltenen Rollenklassen. Diese werden im Kontext der Subteamklasse durch gleichnamige Rollenklassen „implizit“ [Her03c] redefiniert. Das bedeutet, dass im Kontext der Teamklasse `TicTacToe` nur die in ihr enthaltenen Rollenklassen sichtbar sind.

Die genaue Semantik der Teamklassenvererbung wird in Abschnitt 2.2 beschrieben. Das Konzept einer Rollenklasse wird in Abschnitt 2.3 erläutert. Eine Rollenklasse kann sowohl auf Typ-, als auch auf Instanzebene als Attribut eines Teamobjekts betrachtet werden. Sie erbt in Object Teams mehrfach.

2.2 Kapselung von kollaborierenden Rollenklassen in Teamklassen und Teamklassenvererbung

Eine Teamklasse stellt ein instanziierbares Modul dar, das eine Menge von kollaborierenden Rollenklassen enthalten und Verhalten auf Klassenebene definieren kann.

Objekte von Rollenklassen werden analog zu Objekten von inneren Klassen in Java [GJS+00] (§8.1.2) relativ zu einer sie umschließenden Instanz gebildet. Eine Teamklasse stellt also auf Klassen- und auf

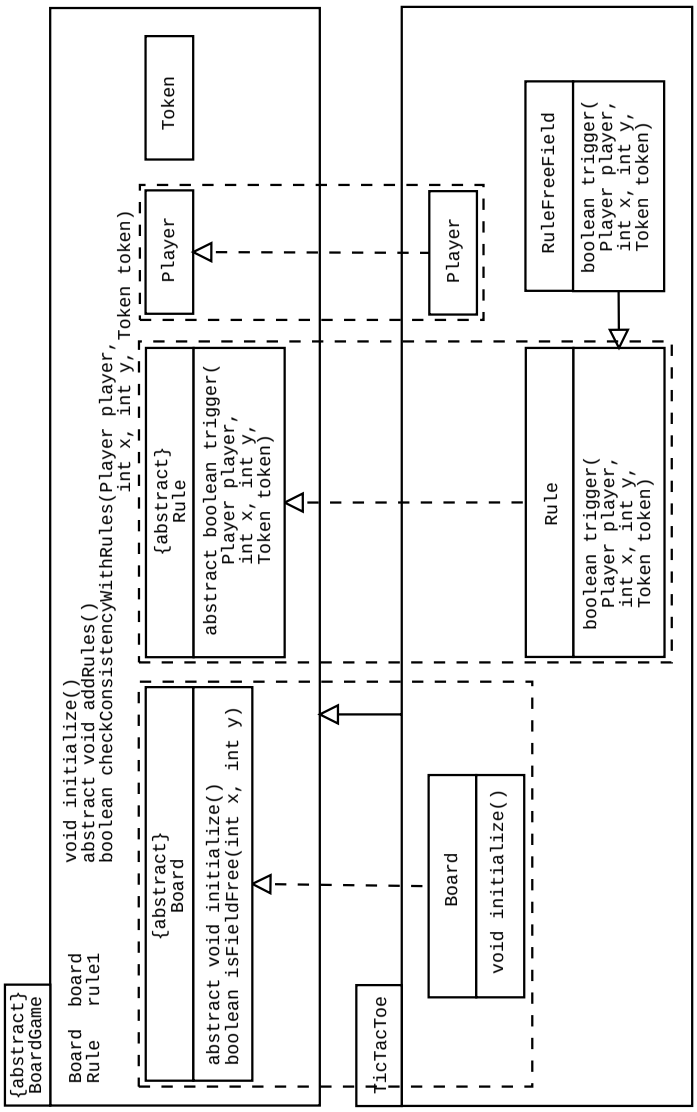

Abbildung 1: Ausschnitt aus der Modellierung eines Rahmenwerkes für Brettspiele mit Object Teams durch die Teamklasse `BoardGame` und einer Instanziierung dieses Rahmenwerkes durch die Subteamklasse `TicTacToe`.

Objektebene einen Container für Rollenklassen und Rollenobjekte dar. So sind zum Beispiel die in Abbildung 1 durch die Attribute `board` und `rule1` referenzierten Rollenobjekte bei Betrachtung einer Instanz der Teamklasse `TicTacToe` in dieser enthalten. Dies wird auch als *Containment* bezeichnet.

Eine Teamklasse stellt ein erbbares Modul dar. Durch die Bildung von Subteamklassen können erstens Attribute und Methoden von Teamklassen wiederverwendet und spezialisiert werden. So verfeinert zum Beispiel die Teamklasse `TicTacToe` die Teamklasse `BoardGame` und implementiert die abstrakten Bestandteile letzterer. Zweitens werden bei der Vererbung zwischen Teamklassen alle Rollenklassen einer Superteamklasse an die entsprechende Subteamklasse vererbt und können dabei analog zum Überschreiben von Methoden über Namensgleichheit redefiniert werden. Das bedeutet, dass durch die Bildung von Subteamklassen kollaborierende Rollenklassen gemeinsam mit dem Kontext, in dem sie enthalten sind, verfeinert werden können.

Wenn durch Rollenklassen gebildete Typen betrachtet werden, die abhängig von der innerhalb von Team- und Rollenklassendeklarationen enthaltenen Referenz auf das aktuelle beziehungsweise umschließende Teamobjekt gebildet werden, dann hat die Redefinition von Rollenklassen beim Bilden von Subteamklassen zur Folge, dass diese Typen durch Rollenklassen des Kontextes der Teamklasse gebunden werden, in dem sie auftreten. So wird zum Beispiel in Abbildung 1 der Typ des Attributs `board` der Teamklasse `BoardGame` durch die Rollenklasse `Board` der Teamklasse `BoardGame` gebildet. In der Subteamklasse `TicTacToe` wird diese Rollenklasse durch die Rollenklasse `Board` der Teamklasse `TicTacToe` überschrieben. Infolgedessen wird beim Zugriff auf das Attribut `board` im Kontext der Teamklasse `TicTacToe` diesem Attributzugriffsausdruck ein durch die Rollenklasse `Board` der Teamklasse `TicTacToe` gebildeter Typ zugewiesen.

Das bedeutet erstens, dass die Redefinition von Rollenklassen beim Bilden einer Subteamklasse zur Folge hat, dass aus Superteamklassen und impliziten Superrollenklassen geerbte Eigenschaften nicht unverändert wiederverwendet werden. Stattdessen werden die in ihnen enthaltenen Deklarationen und Denotationen von durch Rollenklassen gebildeten Typen an die in dem Kontext dieser Subteamklasse sichtbaren Rollenklassen angepasst. Diese Bindung von durch Rollenklassen gebildeten Typen abhängig von dem Kontext der Teamklasse, in dem geerbte Eigenschaften wiederverwendet werden, ist in Abbildung 1 durch Zusammenfassung der namensgleichen Rollenklassen der Superteamklasse `BoardGame` und der Subteamklasse `TicTacToe` visualisiert. Bei der Wiederverwendung der Methode `initialize` der Teamklasse `BoardGame` im Kontext der Teamklasse `TicTacToe` darf zum Beispiel kein Objekt der Rollenklasse `BoardGame.Board` erzeugt werden. Stattdessen muss eine Instanz der Rollenklasse `TicTacToe.Board` gebildet werden, weil die geerbte Rollenklasse `Board` im Kontext der Teamklasse `TicTacToe` „global“ überschrieben ist, und an allen Stellen innerhalb der Teamklasse `TicTacToe` und darin enthaltenen Rollenklassen Objekte vom Typ der Rollenklasse `TicTacToe.Board` erwartet werden.

Zweitens bewirkt die Redefinition von Rollenklassen bei der Bildung einer Subteamklasse, dass die in dieser Subteamklasse deklarierten und denotierten, durch Rollenklassen gebildeten Typen nur durch in dieser Subteamklasse sichtbare Rollenklassen gebildet sind. Dies hat zum Beispiel bei der Implementierung der in der Rollenklasse `BoardGame.Rule` deklarierten Methode `trigger` im Kontext der Subteamklasse `TicTacToe` zur Folge, dass ihre Parametertypen spezialisiert werden. Das bedeutet, dass statisch durch entsprechende Typisierungsregeln und Einschränkungen bezüglich der Wohlgeformtheit von Programmen sichergestellt werden muss, dass innerhalb eines Teamobjektes nur Rollenobjekte referenziert werden, die dessen Anforderungen genügen. Statisch ist bekannt, dass gerade die Rollenobjekte, die in einem Teamobjekt enthalten sind, dessen Anforderungen erfüllen. Diese Aussage wird innerhalb der Ausführungen zur Motivation eines Modells instanzbasierter Typen in Abschnitt 6.1.1 genauer erläutert.

2.3 Rollenklassen und Mehrfacherbung von Rollenklassen

Rollenklassen sind in Teamklassen enthalten. Rollenobjekte sind in Teamobjekten enthalten und existenzabhängig von diesen.

Rollenklassen können bei der Bildung von Subteamklassen redefiniert werden. Das hat auf Typebene zur Folge, dass durch Rollenklassen gebildete Typen, die relativ zu der in Team- und Rollenklassendeklarationen enthaltenen Referenz auf das aktuelle beziehungsweise umschließende Teamobjekt gebildet werden, abhängig von der Teaminstanz, in der sie vorkommen, gebunden werden.

Rollenklassen können mehrfach erben, und zwar erstens entlang einer impliziten Vererbungshierarchie, die über die Namensgleichheit von Rollenklassen voneinander erbender Teamklassen definiert ist. So erbt zum Beispiel die Rollenklasse `Rule` der Teamklasse `TicTacToe` entlang der impliziten Vererbungshierarchie von der Rollenklasse `Rule` der Teamklasse `BoardGame`. Zweitens können Rollenklassen innerhalb einer

Teamklasse entlang der „normalen“ Vererbungshierarchie - der `extends`-Vererbungshierarchie in Java - erben. Letztere etabliert im Gegensatz zur impliziten Vererbungshierarchie eine Subtypbeziehung zwischen Typen, die durch voneinander erbende Rollenklassen gebildet werden, wie zum Beispiel die Rollenklassen `Rule` und `RuleFreeField` der Teamklasse `TicTacToe`. Drittens können Rollenklassen durch ihre Bindung an Klassen von durch Teamklassen adaptierten Anwendungen erben. Dies wird im Rahmen dieser Arbeit nicht betrachtet. Ebenso werden Methodenbindungen nicht betrachtet, da ihre Deklarationen Bindungen von Rollenklassen an Klassen von durch Teamklassen adaptierten Anwendungen voraussetzten.

Verwandte Konzepte von Rollenklassen Das Konzept von Rollenklassen ähnelt dem Konzept der *virtual classes* der Programmiersprache `gbeta`. *Virtual classes* werden zusammen mit Methoden als *virtual patterns* bezeichnet, um folgende Gemeinsamkeiten zum Ausdruck zu bringen. *Virtual patterns* können redefiniert werden und werden dynamisch gebunden. Das Konzept des *family polymorphism* [Ern01] ermöglicht die Kapselung einer Menge - einer Familie - von *virtual classes* und gemeinsame Spezialisierungen dieser Familie. Durch *virtual classes* gebildete Typen werden durch ein *family object* verankert - eine Instanz einer Familie. Ihre exakten Typen können erst bei Kenntniss des dynamischen Typs des *family object* bestimmt werden, relativ zu dem sie verankert sind. Ein entsprechender Ansatz zur instanzbasierten Typisierung von ObjectTeams/Java wird in Abschnitt 6 vorgestellt.

Sowohl Object Teams, als auch das Konzept des *family polymorphism* stellen ein Modulkonzept zur Kapselung einer Menge von kollaborierenden Klassen vor. In Object Teams stellt eine Teamklasse einen Container für Rollenklassen und ein Teamobjekt einen Container für Rollenobjekte dar. In Abschnitt 9.3 wird die Eigenschaft von Object Teams analysiert, auf der Ebene einer Teaminstanz Referenzen auf Rollenobjekte zu kontrollieren. Das Konzept von *confined types* [BV99] führt eine Kontrolle von Referenzen auf Paketebene ein. Durch Annotierungen und Überprüfung von Einschränkungen wird erreicht, dass Instanzen von *confined types* nur im Kontext ihres Pakets referenziert werden können. Einige der in [BV99] spezifizierten Einschränkungen werden bei der Formalisierung von ObjectTeams/Java verwendet, um das *Confinement* von Rollenobjekten zu realisieren. Beispielsweise muss sichergestellt werden, dass auf Rollenobjekten nur in Rollenklassen definierte Methoden ausgeführt werden (siehe Abschnitt 6.3.4). Andere Einschränkungen werden durch Object Teams per se erfüllt wie zum Beispiel, dass Rollenklassen nur durch Rollenklassen derselben Teamklasse verfeinert werden dürfen.

Externalized roles *Externalised roles* stellen Rollenobjekte dar, die nicht nur im Kontext der sie umschließenden Teaminstanz sichtbar sind, sondern uneingeschränkt sichtbar sind. Ihre Typen werden relativ zu Bezeichnern von Variablen verankert, die finale Referenzen auf die Teaminstanzen enthalten, in der die Rollenobjekte enthalten sind. Im Rahmen dieser Arbeit werden nur Rollenobjekte betrachtet, deren Typen relativ zu der in Team- und Rollenklassen enthaltenen Referenz auf das akutelle beziehungsweise umschließende Teamobjekt verankert sind (siehe Abschnitt 6.1.2).

3 Modellierung und Analyse von Typsystemen

Eine typisierte Programmiersprache ermöglicht die Deklaration von Typen in Programmen, die spezifizieren, welche Werte eine Variable bei der Ausführung von Programmen enthalten kann. Ein Typ fasst eine Menge von Werten zusammen [Car97]. Ein Typfehler entsteht dadurch, dass auf einem Element eines Typs eine Funktion angewendet wird, die der Typ nicht enthält, oder durch die Anwendung einer Funktion mit Argumenten, für die sie nicht definiert ist [Ohe01].

Programmiersprachen werden als statisch typisiert bezeichnet, wenn die Abwesenheit von Fehlern beim Kompilieren von Programmen impliziert, dass bei der Ausführung dieser Programme selbst ohne die Durchführung von Laufzeitüberprüfungen keine Typfehler auftreten. Java und infolgedessen auch die Einbettung von Object Teams in Java stellen aufgrund der Durchführung von Laufzeitüberprüfungen zum Beispiel bei der Auswertung von Cast-Ausdrücken keine komplett statisch typisierten Programmiersprachen dar.

Eine Motivation für die Definition eines Typsystems für eine Programmiersprache ist das Verhindern von Typfehlern bei der Ausführung von Programmen dieser Sprache. Weitere Motivationen für die Typisierung von Programmiersprachen und für Typüberprüfungen von Programmen werden in Abschnitt 3.1 vorgestellt. In Abschnitt 3.2 dargestellt wird, welche Schritte für die Formalisierung eines Typsystems einer Programmiersprache nötig sind.

3.1 Motivationen für die Typisierung von Programmiersprachen

Für die Typisierung von Programmiersprachen und die Typüberprüfung von Programmen können folgende methodische Gesichtspunkte angeführt werden.

- Eine vollständige Anreicherung von Schnittstellen um Typinformationen vereinfacht die unabhängige Entwicklung von Software-Moduln [Szy98, Car97].
- Durch Typüberprüfungen während der statischen Analyse von Programmen kann das Finden von Programmierfehlern erleichtert werden [Car97].
- Typinformationen können das Verständnis von Programmen erhöhen und als Dokumentation dienen [Pep97].
- Typisierung kann als analytische Qualitätssicherungsmaßnahme betrachtet werden, da durch die Anreicherung von Programmen um Typinformationen ein Teil von deren Semantik definiert wird, so dass während der statischen Analyse von Programmen diese Typinformationen überprüft und infolgedessen diese Programme teilweise verifiziert werden können.
- Durch die Anreicherung von Programmen um Typinformationen ist ein Überladen von Bezeichnern möglich [Pep97]. Dadurch können dieselben Bezeichner für Operationen mit einander entsprechender Bedeutung verwendet werden.

Weiterhin gilt der folgende Performance-Gesichtspunkt als Motivation für die Typisierung und Typüberprüfung von Programmen.

- Durch die Überprüfung von Typinformationen während der statischen Analyse von Programmen kann oftmals ein besseres Laufzeitverhalten erzielt werden, weil aufwendige Typüberprüfungen zur Laufzeit von Programmen entfallen [Car97].

3.2 Formalisierung von Typsystemen

Ein Typsystem einer Programmiersprache besteht aus einer Menge von Typisierungsregeln. Diese definieren unabhängig von der Implementierung eines bestimmten *Typechecking*-Algorithmus, unter welchen Bedingungen ein Konstrukt einer Programmiersprache korrekt typisiert ist.

Die Formalisierung eines Typsystems einer Programmiersprache ermöglicht es, eine Aussage über die Typzuverlässigkeit dieses Typsystems zu formulieren und zu beweisen. Die Typzuverlässigkeit eines Typsystems einer Programmiersprache impliziert die Typsicherheit der Programmiersprache. Die Eigenschaft einer Programmiersprache, typsicher zu sein, bedeutet, dass das Design der Programmiersprache zusammen mit einer Menge von Einschränkungen bezüglich der Wohlgeformtheit von Programmen das Auftreten von unerwarteten Typfehlern bei der Ausführung von Programmen verhindert [Ohe01]. Ein

erwarteter Typfehler tritt zum Beispiel bei der Auswertung des Cast-Ausdrucks in Java auf, wenn der dynamische Typ des Wertes nicht zum denotierten Typ geweitet werden kann.

Für die Formalisierung eines Typsystems für eine Programmiersprache mit dem Ziel nachzuweisen, dass es typzuverlässig ist, werden in [Car97] fünf Schritte beschrieben.

1. Die Syntax der Programmiersprache ist zu formalisieren.

2. Regeln bezüglich des Geltungsbereichs von Identifiern müssen definiert werden. Dieser ist häufig durch die Syntax der Programmiersprache bestimmt.

3. Ein statisches Environment, relativ zu dem die Typisierungsregeln der Programmiersprache formuliert werden, muss definiert werden, um während der Typisierung eines Programmfragments die Typen von freien Variablen festzustellen.

4. Die Typisierungsregeln der Programmiersprache, die die Beziehungen zwischen Termen und deren statischen Typen definieren, sind zu formalisieren.

   Relationen zwischen Typen müssen definiert werden.

5. Die Semantik der Programmiersprache muss durch die Formalisierung von Auswertungsregeln definiert werden, die die Beziehungen zwischen Termen und Werten zum Ausdruck bringen.

Diese Schritte für die Formalisierung eines Typsystems für ObjectTeams/Java werden ab Kapitel 5 durchlaufen mit dem Ziel, in Kapitel 9 die Typzuverlässigkeit des modellierten Typsystems für ObjectTeams/Java zu analysieren.

Vorher werden im folgenden Kapitel unter anderem das Beweiswerkzeug Isabelle, die Einordnung der Einbettung von ObjectTeams/Java in Isabelle/HOL und der grobe Aufbau dieser Einbettung vorgestellt.

4 Das Beweiswerkzeug Isabelle und Formalisierungen von Java in Isabelle/HOL als Basis für die Einbettung von ObjectTeams/Java in Isabelle/HOL

Das Beweiswerkzeug Isabelle und dessen Instanziierung mit einer Logik höherer Ordnung werden in Abschnitt 4.1 vorgestellt. In Abschnitt 4.2 werden bei der Formalisierung von ObjectTeams/Java in Isabelle/HOL verwendete Konzepte dargestellt. Die Formalisierungen von Java in Isabelle/HOL namens Bali [KNO+02b] und μJava [KNO+02a] werden hinsichtlich der Vorgehensweise bei der Einbettung von Java in Isabelle/HOL in Abschnitt 4.3 vorgestellt. In Abschnitt 4.4 werden die Einordnung der Einbettung von ObjectTeams/Java in Isabelle/HOL und der grobe Aufbau dieser Einbettung dargestellt. Die Konventionen zur Darstellung der Formalisierung von ObjectTeams/Java in Isabelle/HOL in den Kapiteln 5 bis 9 werden in Abschnitt 4.5 vorgestellt.

4.1 Isabelle

Isabelle [Pau94] ist ein generisches, interaktives Beweiswerkzeug, das ein Fragment einer Logik höherer Ordnung (*Higher Order Logic*, HOL), die als Metalogik bezeichnet wird, als Rahmenwerk implementiert. In Isabelle sind durch Instanziierungen dieses Rahmenwerks namens Isabelle/Pure unterschiedliche Logiken eingebettet. Diese werden als Objektlogiken bezeichnet. In dieser Arbeit wird Isabelle/HOL als Beweiswerkzeug verwendet, das die Instanziierung von Isabelle mit der Logik höherer Ordnung von Church [Chu40], der *Simple Theory of Types*, darstellt.

Für die Kodierung der Modellierung und den Nachweis von Aussagen über das Modell wird das Rahmenwerk Isabelle/Isar verwendet. Dieses ermöglicht es, Beweise sowohl in einer neuen Metasprache, als auch in Form von Abfolgen von Taktik-Anwendungen zu denotieren [Wen02]. (Eine Taktik ist eine Funktion zur Veränderung eines Beweiszustands [Pau02a].)

Genau genommen wird also mit Isabelle/Isar/HOL gearbeitet. Im folgenden wird jedoch abkürzend Isabelle/HOL zur Bezeichnung des verwendeten Beweiswerkzeugs und der Einbettung von Churchs Logik höherer Ordnung [Chu40] in Isabelle verwendet.

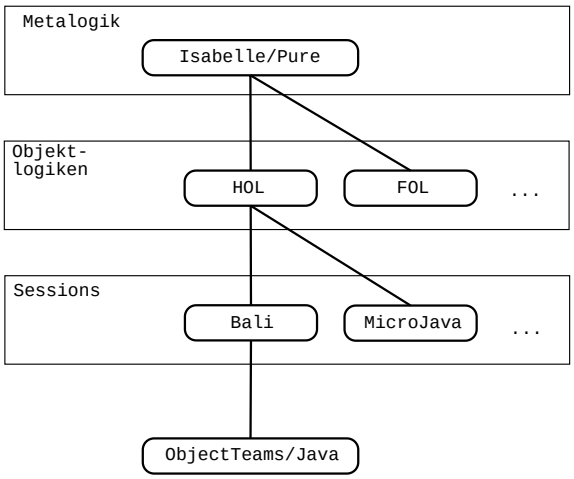

Abbildung 2: Einordnung der Einbettung von ObjectTeams/Java in Isabelle/HOL.

4.2 Modellierungen mit Isabelle/HOL

Bei der Modellierung von ObjectTeams/Java in Isabelle/HOL werden ObjectTeams/Java-Konstrukte und semantische Konzepte von ObjectTeams/Java - wie zum Beispiel die Wohlgeformtheitsbedingungen für ObjectTeams/Java-Programme - auf Entitäten der in Isabelle eingebetteten Logik höherer Ordnung abgebildet. Beispielsweise werden die Wohlgeformtheitsbedingungen für ObjectTeams/Java-Programme durch Definition von Prädikaten formalisiert.

Die in Isabelle eingebettete Logik höherer Ordnung stellt eine typisierte Logik dar. Sie ermöglicht die Modellierung von totalen polymorphen Funktionen höherer Ordnung und Quantifizierungen über Funktionen und Mengen [NOP99].

Formalisierungen mit Isabelle/HOL sind in Moduln organisiert, die als Theorien bezeichnet werden. Eine Theorie besteht aus konzeptionell zusammengehörigen Typ- und Konstantendefinitionen, die die Bestandteile einer Modellierung formalisieren. Desweiteren enthält eine Theorie Aussagen über die formalisierten Modellierungsbestandteile und deren Nachweise. Eine Theorie kann eine andere Theorie erweitern. Die Formalisierung von ObjectTeams/Java in Isabelle/HOL beispielsweise erweitert zwei Theorien aus Bali. Dies ist in Abbildung 2 dargestellt.

Das Typsystem von Isabelle/HOL enthält Basistypen wie zum Beispiel den Typ von Wahrheitswerten, *bool*, Typkonstruktoren wie zum Beispiel den Konstruktor zur Definition von Listen, *list*, Funktionstypen zur Modellierung von totalen Funktionen, $\Rightarrow$, und Typvariablen zur Modellierung von polymorphen Typen, $'a$, [NPW02].

Terme werden in Isabelle/HOL wie im λ-Kalkül mittels Abstraktion und Applikation gebildet.

Die innerhalb einer Theorie formulierten Aussagen können entsprechend ihrer Wichtigkeit als Lemma oder Theorem bezeichnet werden. Sie sind implizit hinsichtlich der in ihnen enthaltenen freien Variablen allquantifiziert und werden mit Hilfe eines Prädikats von der Objektlogik- zur Metalogikebene konvertiert.

Um die Konsistenz von Theorien zu erhalten, sollten Aussagen nicht als Axiome formuliert werden, sondern aus Modellierungen abgeleitet werden [NOP99]. Auf Typebene muss bei der Definition eines neuen Typs entweder automatisch oder per Hand gezeigt werden, dass dieser nicht leer ist. Typdefinitionen werden bei der Formalisierung von ObjectTeams/Java in Isabelle/HOL beispielsweise verwendet, um Namen von Typen in ObjectTeams/Java oder das Konzept von Team- und Rollenklassen zu formalisieren.

Ein konkretes Beispiel für eine Theorie ist im Anhang enthalten. Dabei handelt es sich um die Kodierung des in Abbildung 1 dargestellten Object Teams-Beispiels.

4.3 Formalisierungen von Java in Isabelle/HOL

Die Programmiersprache Java [GJS+00] wurde mit dem Fokus des Nachweises ihrer Typsicherheit in Isabelle/HOL eingebettet. Die Formalisierungen von Java namens Bali [KNO+02b] und μJava [KNO+02a] unterscheiden sich hinsichtlich des behandelten Sprachumfangs. Die in dieser Arbeit beschriebene Einbettung von ObjectTeams/Java in Isabelle/HOL ist an diese Formalisierungen angelehnt und baut auf zwei Theorien aus Bali auf.

Entsprechend der Formalisierung der Programmiersprache Java in [KNO+02a] und [KNO+02b] handelt es sich bei der Einbettung von ObjectTeams/Java in Isabelle/HOL um eine tiefe Einbettung. Das bedeutet, dass die syntaktischen Konstrukte von ObjectTeams/Java durch Elemente von Isabelle/HOL formalisiert werden, und dass die Semantik der syntaktischen Konstrukte von ObjectTeams/Java auf der Basis dieser Formalisierungen modelliert wird. Bei einer flachen Einbettung einer Programmiersprache in eine Logik wird die Semantik der Konstrukte der Programmiersprache direkt durch Entitäten der Logik formalisiert. Eine Motivation für eine tiefe Einbettung von Java in Isabelle/HOL war die Formulierung metatheoretischer Aussagen über die Sprache selbst, beispielsweise über die Typsicherheit von Java [Ohe01]. Für ObjectTeams/Java wird neben der Betrachtung einer Typzuverlässigkeitsaussage für ein Typsystem von ObjectTeams/Java in Abschnitt 9.2 eine metatheoretische Aussage über die Kontrolle von Referenzen auf Rollenobjekte, die das Teams-Konstrukt ausübt, in Abschnitt 9.3 formuliert.

4.4 Überblick über die Einbettung von ObjectTeams/Java in Isabelle/HOL

Die Einbettung von ObjectTeams/Java in Isabelle/HOL ist in Abbildung 3 dargestellt. Sie setzt sich aus folgenden Bestandteilen zusammen.

- Eine abstrakte Syntax für ObjectTeams/Java wird in Kapitel 5 formalisiert. Die Konzepte von Team- und Rollenklassen werden modelliert. Neben der `extends`-Vererbungsbeziehung zwischen Klassen wird die implizite Vererbungsbeziehung zwischen Rollenklassen formalisiert.

- Aufbauend auf der Formalisierung dieser abstrakten Syntax für ObjectTeams/Java wird in Kapitel 6 ein Typsystem für ObjectTeams/Java modelliert. Es wird ein Modell statischer und dynamischer instanzbasierter Typen motiviert und dargestellt. Relationen zwischen statischen und dynamischen instanzbasierten Typen werden definiert. Regeln zur Typisierung von ObjectTeams/Java-

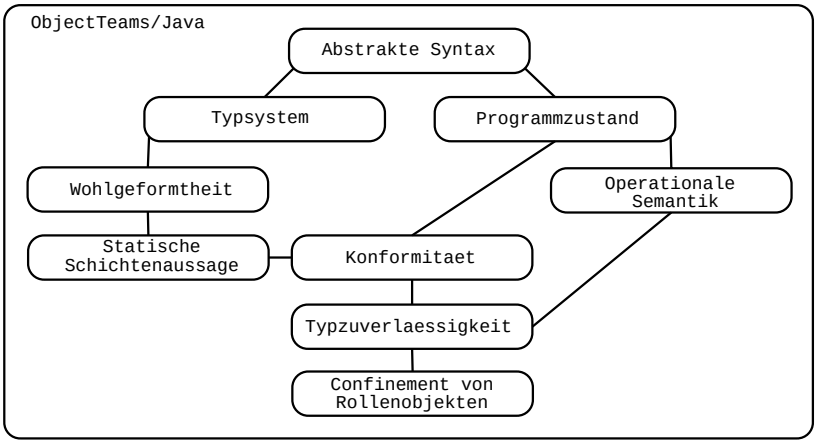

Abbildung 3: Grober Aufbau der Einbettung von ObjectTeams/Java in Isabelle/HOL.

Konstrukten werden modelliert, die die Bindung von Rollentypen relativ zu dem Teamkontext, in dem sie vorkommen, realisieren.

- In Kapitel 7 werden Wohlgeformtheitsbedingungen für ObjectTeams/Java-Programme spezifiziert. Desweiteren wird eine statische Schichtenaussage bewiesen. Diese bringt zum Ausdruck, dass die bei der Typisierung von wohlgeformten ObjectTeams/Java-Programmen abgeleiteten Typen wohlkonstruiert sind, und durch Rollenklassen gebildete Typen abhängig von dem Kontext der Teamklasse, in dem sie auftreten, gebunden werden.
- Basierend auf der Formalisierung der abstrakten Syntax für ObjectTeams/Java wird in Kapitel 8 ein Programmzustand und eine operationale Semantik zur Auswertung von ObjectTeams/Java-Konstrukten modelliert.
- Mit dem Ziel, eine Typzuverlässigkeitsaussage bezüglich des in Kapitel 6 modellierten Typsystems für ObjectTeams/Java zu formulieren, werden in Kapitel 9 Prädikate über die Konformität zwischen dynamischen Typen von bei der Auswertung von Termen produzierten Werten und den statischen Typen dieser Terme modelliert. Die Typzuverlässigkeit des in Kapitel 6 modellierten Typsystems für ObjectTeams/Java wird betrachtet. Eine Aussage über das *Confinement* von Rollenobjekten wird formuliert und unter der Voraussetzung, dass die Typzuverlässigkeitsaussage vollständig nachgewiesen werden kann, bewiesen.

4.5 Konventionen zur Darstellung der Formalisierung

In den Kapiteln 5 bis 9 wird die Formalisierung von ObjectTeams/Java in Isabelle/HOL präsentiert. Dabei werden folgende Konventionen zur Darstellung der Formalisierung verwendet. Diese werden anhand der folgenden Typdefinition und Konstantendeklaration und -definition aus der Formalisierung von ObjectTeams/Java in Isabelle/HOL erläutert.

Beispiel für eine Typdefinition:
`datatype` *tname*
= Object
| Team
| SXcpt *xcpt*
| TName *tnam*

Beispiel für eine Konstantendeklaration:
`consts`
bclass :: $prog \Rightarrow (tname, class)table$

Der Typ $('a,'b)table$ modelliert *lookup table* [KNO+02b] (Table.thy). Er ist als Typsynonym für den Typ $'a \Rightarrow 'b\ option$ definiert.

Beispiel für eine Konstantendefinition:

```
defs
  bclass_def:
  bclass G C == table_of (bclass_decls G) C
```

- Namen von Typen werden *kursiv* notiert.

  Beispielsweise formalisiert der Typ *tname* Namen von ObjectTeams/Java-Typen.

  Bei der Deklaration einer Konstante werden der Name der Konstante und ihr Typ durch :: getrennt.

- Schlüsselwörter von Isabelle/HOL werden in `Courier` notiert.

  Das Schlüsselwort `datatype` kennzeichnet die Definition des Typs *tname* mit Hilfe des `datatype`-Definitionspakets von Isabelle/HOL.

  Die Schlüsselwörter `consts` und `defs` dienen zur Deklaration und Definition der Konstanten bclass.

- Konstruktoren von Typen, die mit Hilfe des `datatype`-Definitionspakets definiert sind, werden in Sansserif notiert, wie zum Beispiel der Konstruktor Object zur Formalisierung des Namens der Java-Standardklasse `Object`. (Innerhalb der Beschreibungen werden ObjectTeams/Java- und Java-Schlüsselwörter und -Standardklassennammen in `Courier` dargestellt.)

  Desweiteren werden alle Bestandteile von Konstantendefinitionen und Lemmata in Sansserif notiert.

Eine Einführung in Isabelle/HOL ist in [NPW02] enthalten.

Die vollständige Formalisierung kann bei Anfrage zugesandt werden (siehe E-Mail-Adresse auf der Titelseite).

5 Modellierung einer abstrakten Syntax für ObjectTeams/Java

Die Bestandteile einer abstrakten Syntax für ObjectTeams/Java werden modelliert. In Abschnitt 5.1 werden in ObjectTeams/Java-Programmen vorkommende Typ-, Methoden- und Variablennamen formalisiert. Desweitern wird der Name einer modellierungsspezifischen Variablen formalisiert. Diese bezeichnet den Teamkontext, mit dem die Regeln zur Typisierung und die Regeln zur Auswertung von ObjectTeams/Java-Konstrukten parametrisiert sind. In Abschnitt 5.2 ist die Modellierung von Anweisungen und Ausdrücken enthalten. In Abschnitt 5.3 wird der strukturelle Aufbau von ObjectTeams/Java-Programmen formalisiert. Team- und Rollenklassen werden als ausgezeichnete Klassentypen modelliert. Abstrakte Methoden und Klassen werden formalisiert. In Abschnitt 5.4 werden die `extends`-Vererbungsrelation zwischen Klassen und die implizite Vererbungsrelation zwischen Rollenklassen formalisiert. Auf der Basis dieser Vererbungsrelationen werden Funktionen zur Traversierung der durch die `extends`- und die implizite Vererbungsrelation etablierten Klassenhierarchien definiert. Diese Traversierungsfunktionen werden bei der Modellierung des Zugriffs auf Attribute und Methoden von Klassen verwendet. Zur Definition dieser Traversierungsfunktionen ist der Nachweis der Wohlfundiertheit der Vererbungsrelationen erforderlich. Das bedeutet, dass gezeigt werden muss, dass die Vererbungsrelationen endlich und azyklisch sind. In Abschnitt 5.5 werden Prädikate bezüglich der Wohlstruktuiertheit von Klassendeklarationen und Programmen definiert. Diese Prädikte dienen zum einen der Überprüfung, ob die `extends`- und die implizite Vererbungsrelation azyklisch sind. Zum anderern modellieren sie strukturelle Einschränkungen hinsichtlich von Klassendeklarationen. In Abschnitt 5.6 wird nachgewiesen, dass die `extends`- und die implizite Vererbungsrelation wohlfundiert sind. Es wird bewiesen, dass sie endlich und azyklisch sind. In Abschnitt 5.7 werden Induktionsschemata über die `extends`- und die implizite Vererbungsrelation aufgestellt. Diese Induktionsschemata werden zum Beispiel zum Nachweis der statischen Schichtenaussage in Abschnitt 7.2 verwendet. In Abschnitt 5.8 werden auf der Basis der `extends`- und der impliziten Vererbungsrelation Funktionen zur Traversierung der durch die `extends`- und die implizite Vererbungsrelation etablierten Klassenhierarchien definiert.

Die Modellierung der Java-Bestandteile einer abstrakten Syntax für ObjectTeams/Java ist an die Formalisierungen von Java in Isabelle/HOL namens Bali [KNO+02a] und μJava [KNO+02b] angelehnt und erweitert diese. Die Formalisierung des Exception-Konzepts von Java ist ohne Veränderung aus [KNO+02a] übernommen.

5.1 Modellierung von Namen

Es werden Namen von ObjectTeams/Java-Typen durch den Typ *tnam*, Methodennamen durch den Typ *mname* und Variablennamen durch den Typ *vnam* modelliert. Die Typen *tnam*, *mname* und *vnam* werden unter Verwendung des Schlüsselworts `typedecl` deklariert. Außer der Zusicherung, dass die Typen nicht leer sind, werden nur die Namen der Typen eingeführt, ohne sie weiter zu definieren. Die Typen könnten als Parameter der Theorie aufgefasst werden [NPW02].

`typedecl` *tnam*
`typedecl` *mname*
`typedecl` *vnam*

Der Typ *xcpt* dient zur Modellierung von Exception-Klassennamen. Diese werden als Konstruktoren des Typs *xcpt* formalisiert. Der Typ *xcpt* ist unter Verwendung des Schlüsselwortes `datatype` definiert. Mit Hilfe des `datatype`-Definitionspakets können rekursive, gegenseitig rekursive und indirekt oder geschachtelt rekursive Typen modelliert werden. (Ein Beispiel für die Definition von gegenseitig und indirekt rekursiven Typen ist in Abschnitt 5.2 enthalten.) Bei der Definition eines Typs unter Verwendung des `datatype`-Definitionspakets generiert Isabelle/HOL automatisch strukturelle Induktionsregeln, und es werden beispielsweise Theoreme über die „Verschiedenheit“ (*freeness*) der Konstruktoren des Typs bewiesen. Funktionen über mit dem `datatype`-Paket definierten Typen können mit primitiver Rekursion unter Verwendung des Schlüsselwortes `primrec` definiert werden.

`datatype` *xcpt*
= NullPointer
| OutOfMemory
| ClassCast

Namen von Java- und ObjectTeams/Java-Standardklassen und benutzerdefinierte Namen von Typen in

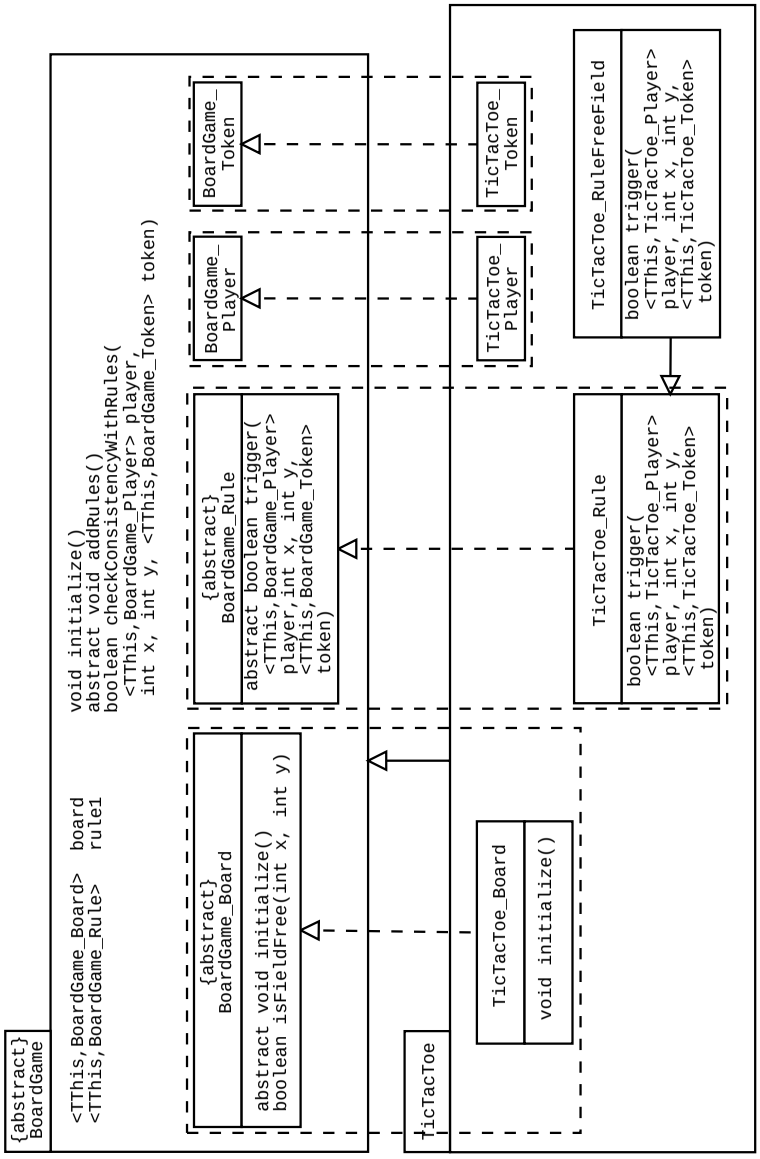

Abbildung 4: Ausschnitt aus der Modellierung des Brettspiel-Rahmenwerks mit flachem Typnamensraum, Redefinition aller Rollenklassen bei der Bildung von Subteamklassen und instanzbasierten Typen.

ObjectTeams/Java sind durch den Typ *tname* zusammengefasst. Der Konstruktor Object modelliert den Namen der Java-Standardklasse `Object`, die die Superklasse aller Klassen in Java darstellt. Analog dazu ist der Typ *tname* um den Konstruktor Team zur Modellierung des Namens der Klasse `Team` erweitert. Die Klasse `Team` stellt die Superklasse aller Teamklassen in ObjectTeams/Java dar [Her03c]. Vereinfachend ist kein hierarchischer Typnamensraum modelliert.

`datatype` *tname*
= Object
| Team
| SXcpt *xcpt*
| TName *tnam*

Es wird davon ausgegangen, dass bei der Kodierung von ObjectTeams/Java-Programmen alle qualifizierten Typnamen auf eindeutige flache Typnamen abgebildet werden. Das Resultat der Transformationen zur Kodierung des in Abbildung 1 vorgestellten Ausschnitts aus einer Object Teams-Anwendung ist in Abbildung 4 und im Anhang dargestellt. Die Formalisierung eines hierarchischen Typnamensraums würde eine realitätsnähere Modellierung von Typnamen von in Teamklassen enthaltenen Rollenklassen und geschachtelten Teamklassen ermöglichen. In [KNO+02b] werden Typnamen formalisiert, die mit einem Paketnamen qualifiziert sind. Diese Modellierung ist zur Darstellung von beliebig tiefen Qualifizierungen von Typnamen nicht ausreichend. Die durch die Modellierung eines flachen Typnamensraumes verlorengehenden Informationen zum Beispiel über das Enthaltensein von Rollenklassen in Teamklassen werden als strukturelle Informationen von Team- und Rollenklassentypen formalisiert. Team- und Rollenklassentypen werden in Abschnitt 5.3 modelliert. Diese strukturellen Informationen werden innerhalb von Wohlstrukturiertheitsprädikaten überprüft, die in Abschnitt 5.5 definiert werden.

Namen von lokalen Variablen, Methodenparametern und Attributen und zwei ausgezeichnete Variablennamen werden durch den Typ *vname* zusammengefasst. Der Konstruktor VName modelliert Namen von lokalen Variablen, Methodenparametern und Attributen. Der Name des Java-Schlüsselworts `this` zur Bezeichnung der in Instanzmethoden und Konstruktoren enthaltenen Referenz auf das aktuelle Objekt, auf dem eine Methode ausgeführt wird, oder das gerade konstruiert wird, ist durch den Konstruktor This formalisiert. Die Bedeutung des Konstruktors TContext wird im folgenden in Abschnitt 5.1.1 erläutert.

`datatype` *vname*
= VName *vnam*
| This
| TContext

5.1.1 Modellierungsspezifische Variable TContext

Der Konstruktor TContext des Typs *vname* dient zur Modellierung des Namens einer modellierungsspezifischen Variablen. Sie ist nicht Bestandteile der Syntax von ObjectTeams/Java. Die modellierungsspezifische Variable TContext repräsentiert den Teamkontext, mit dem die Regeln zur Typisierung und die Regeln zur Auswertung von ObjectTeams/Java-Konstrukten parametrisiert werden. Während der statischen Analyse ist der statische Typ dieser Variablen der Teamkontext, der zur Typisierung von ObjectTeams/Java-Konstrukten verwendet wird. Im folgenden wird dieser auch als statischer Teamkontext bezeichnet. Er dient zur Eingrenzung von durch Rollenklassen gebildeten Typen von ObjectTeams/Java-Konstrukten. Bei der Ausführung von ObjectTeams/Java-Programmen enthält die Variable TContext eine Referenz auf das Teamobjekt, durch dessen dynamischen Typ die tatsächlichen Typen von ObjectTeams/Java-Konstrukten bestimmt werden. Der dynamische Typ des durch diese Variablen referenzierten Teamobjekts wird auch als dynamischer Teamkontext bezeichnet. Der Teamkontext, der in einem statischen Environment enthalten ist, wird im folgenden auch als aktueller Teamkontext bezeichnet.

5.2 Modellierung von Termen

Es werden in Methodenkörpern enthaltene Anweisungen und Ausdrücke als gegenseitig und indirekt rekursive Typen *stmt*, *expr* und *target* unter Verwendung des `datatype`-Pakets formalisiert. Neben der Modellierung der Typen *expr* und *stmt* zur Formalisierung von Ausdrücken und Anweisungen durch Definition entsprechender Konstruktoren ist der Typ *target* zur Konstruktion von Ausdrücken definiert, auf denen Methoden aufgerufen werden können. Der Typ *target* wird modelliert, um das neue

ObjectTeams/Java-Schlüsselwort `tsuper` [Her03c] nur zum Aufruf von entlang der impliziten Vererbungshierarchie redefinierten Methoden verwenden zu können. Nach der Definition der Typen werden sie detaillierter beschrieben. Bei der Definition der Typen werden syntaktische Annotationen für einige der Konstruktoren zur besseren Lesbarkeit definiert.

```
datatype expr
    = NewC ic tname
    | Cast ty expr
    | Lit val
    | Inst expr ref_ty
    | Call tname inv_mode target mname (expr list)        ({_,_}_._'(_') [10,10,90,99,10]90)
    | LAcc vname
    | LAss vname expr                                     (_::=_ [90,90]90)
    | FAcc tname expr vname                               ({_}_._ [10,90,99]90)
    | FAss tname expr vname expr                          ({_}_._:=_ [10,90,99,90]90)
    | Tthis

and target
    = Super
    | TSuper
    | TargetExpr expr

and stmt
    = Skip
    | Expr expr
    | Comp stmt stmt                                      (_;; _ [61,60]60)
    | Cond expr stmt stmt                                 (If '(_') _ Else _ [80,79,79]70)
    | Loop expr stmt                                      (While '(_')_ [80,79]70)
```

Der Typ *stmt* modelliert Anweisungen und ist aus [KNO+02a] übernommen. Er definiert Konstruktoren zur Formalisierung einer leeren Anweisung (Skip), der Konvertierung eines Ausdrucks vom Typ *expr* in eine Anweisung (Expr) und der Komposition von Anweisungen (Comp). Der Konstruktor Cond modelliert konditionale Anweisungen und der Konstruktor Loop Schleifenanweisungen.

Der Typ *expr* modelliert Ausdrücke und definiert Konstruktoren zur Formalisierung von Cast-Ausdrükken (Cast) und von `instanceof`-Ausdrücken (Inst). (Der Typ *ref_ty* modelliert instanzbasierte Referenztypen. Primitive Typen und instanzbasierte Referenztypen sind durch den Typ *ty* zusammengefasst. Die Definitionen dieser Typen sind in Abschnitt 6.1.3 enthalten.) Literale werden durch den Konstruktor Lit modelliert, der ein Argument vom Typ *val* erhält. (Der Typ *val* modelliert Werte und wird am Ende dieses Abschnitts definiert.) Die Konstruktoren LAcc und LAss modellieren den Zugriff auf Variablen des lokalen Environments beziehungsweise die Zuweisung von Ausdrücken an Variablen des lokalen Environments. Die Zuweisung von Ausdrücken an Attribute wird durch den Konstruktor FAss modelliert.

Die Konstruktoren NewC, FAcc, Call und Tthis des Typs *expr* und die Konstruktoren des Typs *target* werden im folgenden genauer beschrieben.

- NewC. Bei der Bildung einer Instanz einer Klasse muss neben dem Namen der Klasse, von der ein Objekt gebildet werden soll, der Instanzkontext angegeben werden, relativ zu dem das Objekt erzeugt werden soll. Es wird davon ausgegangen, dass bei der Formalisierung von Methodendeklarationen innerhalb eines Vorverarbeitungsprozesses dieser Instanzkontext nach dem `new`-Operator eingefügt wird.

  Der Typ *ic* modelliert diesen Instanzkontext und ist in Abschnitt 6.1.3 definiert. Bezogen auf die Instanziierung einer Klasse bringt er zum Ausdruck, abhängig von welchem umschließenden Objekt eine Instanz einer Klasse gebildet wird - gegebenenfalls unabhängig von einem umschließenden Objekt.

- Call. Vereinfachend wird im Gegensatz zu [KNO+02a] und [KNO+02b] kein Überladen von Methoden betrachtet, weshalb bei der Modellierung eines Methodenaufrufausdrucks unter Verwendung des Konstruktors Call diesem keine Parametertypen als Argument übergeben werden.

  Bei der Formalisierung von Methodenaufrufausdrücken muss innerhalb eines Vorverarbeitungsprozesses der Aufrufmodus einer Methode eingefügt werden. Er dient zur Realisierung unterschiedlicher

Auswertungsstrategien beim Aufruf von Methoden und ist durch den Typ *inv_mode* formalisiert. Die Definition des Typs *inv_mode* erfolgt nach dieser Aufzählung.

- Call und FAcc. Die Ausdrücke Call und FAcc modellieren Methodenaufrufe und Attributzugriffe. Sie sind um sogenannte Typannotationen angereichert. Eine Typannotation stellt den Namen der Klasse dar, in der die Eigenschaft, auf die zugegriffen wird, definiert ist. Typannotationen dienen zur Realisierung von Aufrufen redefinierter und statischer Methoden und zur Realisierung des statischen Bindens von Attributen. In Anlehnung an [ON99] wird davon ausgegangen, dass diese Typannotationen innerhalb eines Vorverarbeitungsprozesses eingefügt werden. Sie werden innerhalb der Typisierungsregeln überprüft, die in Abschnitt 6.3.4 definiert werden.

  Vereinfachend wird davon ausgegangen, dass Attributzugriffe und Methodenaufrufe immer qualifiziert stattfinden.

- Tthis. Der Ausdruck Tthis formalisiert die in Rollenklassendeklarationen enthaltene Referenz auf das aktuelle umschließende Teamobjekt. Er abstrahiert von der Java-Notation `OuterClass.this`.

  Tthis stellt im Gegensatz zu der modellierungsspezifischen Variablen TContext (siehe Abschnitt 5.1.1) einen Ausdruck dar, der von der konkreten ObjectTeams/Java-Syntax abstrahiert. Er ist im Gegensatz zur Variablen TContext Bestandteil von in Isabelle/HOL kodierten ObjectTeams/Java-Programmen.

  Der Variablen TContext wird immer der speziellste bekannte Teamkontext zugewiesen. Der Typ des Ausdrucks Tthis hingegen ist durch die Teamklasse, in der der Ausdruck definiert ist, bestimmt (siehe Abschnitt 6.3.4).

- Super. In Analogie zur Verwendung des Schlüsselwortes `tsuper` soll `super` nur zum Zugriff auf entlang der `extends`-Vererbungshierachie redefinierte Methoden verwendet werden können. Dies stellt eine Einschränkung der Verwendung des Java-Schlüsselwortes `super` dar, da `super` in Java auch zum Zugriff auf überdeckte Superklassenattribute verwendet werden kann [GJS+00] (§ 15.11.2). Dieser Zugriff kann jedoch durch einen Cast des Typs von `this` zum Typ der Superklasse simuliert werden.

  Desweiteren sollen direkte Zugriffe auf überdeckte Attribute und Aufrufe redefinierter Methoden einer umschließenden Klasse unterbunden werden. Ein Ausdruck der Form `OuterClass.super.feature` [GJS+00] (§ 15.11.2) soll nicht realisierbar sein.

- TargetExpr. Dieser Konstruktor ermöglicht die Umwandlung eines beliebigen Ausdrucks vom Typ *expr* in einen Ausdruck vom Typ *target*, so dass jeder Ausdruck vom Typ *expr* als Ausdruck verwendet werden kann, auf dem Methoden aufgerufen werden.

Der Typ *inv_mode* modelliert mögliche Modi beim Aufruf von Methoden. Diese werden bei der Auswertung von Methodenaufrufen zur Laufzeit verwendet [GJS+00] (§15.12.3). Die möglichen Modi beim Aufruf von Methoden sind durch die Konstruktoren des Typs *inv_mode* formalisiert. Wird eine Methode auf dem Ausdruck Super aufgerufen, dann ist der Aufrufmodus der Methode SuperM. Wird eine Methode auf dem Ausdruck TSuper aufgerufen, dann ist der Aufrufmodus der Methode TSuperM. Wird eine statische Methode aufgerufen, dann ist der Aufrufmodus der Methode Static. Ansonsten handelt es sich um den Aufruf einer Instanzmethode. Dann ist der Aufrufmodus der Methode IntVir.

`datatype` *inv_mode*
= SuperM
| TSuperM
| Static
| IntVir

Das Java-Schlüsselwort `this` wird als syntaktische Abkürzung unter Verwendung der Schlüsselwörter `syntax` und `translations` modelliert. Diese Definition ist aus [KNO+02b] übernommen.

`syntax`
this :: *expr*
`translations`
this == LAcc This

Der Typ *term* fasst die Typen *expr*, *stmt* und *target* und den Typ *expr list*, der eine Liste von Ausdrücken modelliert, zusammen. Durch das Schlüsselwort `types` wird kein neuer Typ definiert, sondern lediglich ein Typsynonym als syntaktische Abkürzung für den rechts des Gleichheitszeichens stehenden Typ eingeführt [Wen02].

`types` *term* = (*stmt*, *expr*, *target*)`sum3` + *expr list*)

Beispiele für die Kodierung von Methodenanweisungen sind im Anhang enthalten. Die Methoden des in Abbildung 1 vorgestellten Ausschnittes aus dem Brettspiel-Rahmenwerk werden im Anhang mit Object-Teams/Java implementiert und in Isabelle/HOL kodiert.

Werte Der Typ *val* modelliert Werte. Die Konstanten the_Bool, the_Intg und the_Addr sind mittels primitiver Rekursion über den Typ *val* definiert und modellieren den Zugriff auf den entsprechenden Wert. Sie stellen *underdefined* Funktionen [NPW02] dar, da sie nicht für alle Konstruktoren des Typs *val* Gleichungen definieren. Die Konstanten werden bei der Definition der Auswertungsregeln für ObjectTeams/Java-Konstrukte in Abschnitt 8.2 verwendet. Der Typ *loc* modelliert Referenzen auf Objekte und wird in Abschnitt 6.1.4 definiert.

```
datatype val
    = Unit
    | Bool bool
    | Intg int
    | Null
    | Addr loc

consts
    the_Bool :: val ⇒ bool
    the_Intg :: val ⇒ int
    the_Addr :: val ⇒ loc

primrec
    the_Bool (Bool b) = b

primrec
    the_Intg (Intg i) = i

primrec
    the_Addr (Addr a) = a
```

Das Prädikat is_reference überprüft, ob es sich bei seinem Argument vom Typ *val* um eine Referenz auf ein Objekt handelt.

```
consts
    is_reference :: val ⇒ bool

primrec
    is_reference_Unit: is_reference (Unit) = False
    is_reference_Bool: is_reference (Bool b) = False
    is_reference_Intg: is_reference (Intg i) = False
    is_reference_Null: is_reference (Null) = False
    is_reference_Addr: is_reference (Addr a) = True
```

5.3 Modellierung von Programmen

Im folgenden werden strukturelle Bestandteile von Programmen und Programme selbst unter Verwendung von Typsynonymen und Record-Typdefinitionen mit Hilfe des Schlüsselwortes `record` modelliert. Die Formalisierung von Programmbestandteilen als Record-Typen ermöglicht die Modellierung von benannten Komponenten von Tupeltypen. Dabei werden automatisch Selektor- und Aktualisierungsfunktionen von Isabelle/HOL generiert werden. Record-Typen können erweitert werden. Beispielsweise wird der Typ *abs_member* durch den Typ *mhead* und den Typ *cbody* erweitert. Beide erweiterten Typen besitzen ein Feld namens abstract, auf das mittels einer entsprechend dem Feldnamen benannten Selektorfunktion zugegriffen werden kann.

Der Record-Typ *abs_member* formalisiert den Java-Modifier `abstract` zur Kennzeichnung von abstrakten Methoden und Klassen.

`record` *abs_member* =
  abstract :: *bool*

Attributdeklarationen sind durch das Typsynonym *fdecl* formalisiert. Eine Attributdeklaration besteht aus dem Namen des Attributs vom Typ *vname* und einer Komponente vom Typ *field* besteht. Der Typ *field* ist als Record-Typ definiert und enthält in der jetzigen Modellierung nur ein Feld namens field_ty zur Modellierung des Typs eines Attributs. Er ermöglicht das Hinzufügen weiterer Felder beispielsweise zur Formalisierung von finalen Attributen [GJS+00] (§8.3.1.2), die bei der Formalisierung von *externalized roles* (siehe Abschnitt 2.3) benötigt werden. Diese werden im Rahmen dieser Arbeit nicht behandelt.

`types` *fdecl* = *vname* ∗ *field*

`record` *field* =
  field_ty :: *ty*

Methodensignaturen sind durch den Record-Typ *sig* modelliert. Eine Methodensignatur besteht aus dem Namen der Methode und der Liste der Methodenparametertypen [GJS+00] (§8.4.2).

`record` *sig* =
  name :: *mname*
  param_tys :: *ty list*

Methodenköpfe sind durch den Record-Typ *mhead* modelliert. Ein Methodenkopf besteht aus Methoden-Modifiern, dem Resultattyp der Methode und der Liste der Methodenparameternamen. In dieser Modellierung sind nur die Methoden-Modifier `static` und `abstract` formalisiert.

`record` *mhead* = *abs_member* +
  static :: *bool*
  res_ty :: *ty*
  params :: *vname list*

Methodenkörper sind durch den Record-Typ *mbody* modelliert. Ein Methodenkörper besteht aus einer Liste von lokalen Variablen, die durch ihren Namen und Typ ausgezeichnet sind. Er enthält einen Block von Anweisungen vom Typ *stmt*. Vereinfachend wird genau ein Methodenrückgabeausdruck einer Methode modelliert. Das hat zur Folge, dass Methodendeklarationen bei ihrer Kodierung in Isabelle/HOL dementsprechend transformiert werden müssen. Der Vergleich der Implementierung der Methode `trigger` der Rollenklasse `RuleFreeField` (erhältlich unter http://www.objectteams.org) und der im Anhang enthaltenen Kodierung dieser Methode in Isabelle/HOL veranschaulicht den erforderlichen Transformationsprozess.

`record` *mbody* =
  lcl_vars :: (*vname* ∗ *ty*) *list*
  stmt :: *stmt*
  res :: *expr*

Methodendefinitionen sind durch den Record-Typ *methd* modelliert. Dieser erweitert den Typ *mhead* und definiert ein Feld namens body zur Modellierung eines optionalen Methodenkörpers durch Einpacken des Typs *mbody* in den Typ *option*. Dadurch wird die Formalisierung von abstrakten Methoden ermöglicht, die keinen Methodenkörper definieren.

`record` *methd* = *mhead* +
  body :: *mbody option*

Methodendeklarationen werden durch das Typsynonym *mdecl* modelliert. Eine Methodendeklaration besteht aus einer Methodensignatur und einer Methodendefinition.

`types` *mdecl* = *sig* ∗ *methd*

Der Record-Typ *cbody* modelliert Klassenkörper bestehend aus einer Liste von Attribut- und Methodendeklarationen. Er erweitert den Record-Typ *abs_member*, um die Deklaration von abstrakten Klassen zu modellieren.

record $cbody = abs\_member +$
  attrs :: $fdecl\ list$
  methods :: $mdecl\ list$

Basisklassendefinitionen werden durch den Typ *class* modelliert. Dieser erweitert den Typ *cbody* und definiert ein Feld namens super zur Formalisierung der Superklasse einer Klasse entlang der `extends`-Vererbunghierarchie. Der Typ *class* wird durch den Typ *rclass* zur Formalisierung von Rollenklassendefinitionen und den Typ *tclass* zur Darstellung von Teamklassendefinitionen erweitert. Die Typen *rclass* und *tclass* definieren Felder zur Modellierung strukutureller Klassendefinitionsinformationen, die durch die Verwendung eines flachen Typnamensraums erforderlich sind. Der Typ *rclass* enthält ein Feld namens implicit zur Formalisierung der impliziten Superrollenklasse einer Rollenklasse - gegebenenfalls keiner - und ein Feld namens encl_class zur Modellierung des Enthaltenseins einer Rollenklasse in einer Teamklasse. Der Typ *tclass* enthält ein Feld namens roles zur Modellierung der in einer Teamklasse enthaltenen Rollenklassen. Die Definition eines Rollenklassentyps ermöglicht eine einfache Erweiterung der in dieser Arbeit vorgestellten Modellierung um Bindungen von Rollenklassen und Methodenbindungen (siehe Abschnitt 2.3).

record $class = cbody +$
  super :: $tname$

record $rclass = class +$
  implicit :: $tname\ option$
  encl_class :: $tname$

record $tclass = class +$
  roles :: $tname\ list$

Klassendeklarationen werden als Typen *bclass_decl*, *tclass_decl* und *rclass_decl* bestehend aus einem Klassennamen vom Typ *tname* und dem entsprechenden Klassendefinitionstyp modelliert.

types $bclass\_decl = tname * class$
types $tclass\_decl = tname * tclass$
types $rclass\_decl = tname * rclass$

Programme werden als Elemente des Typs *prog* formalisiert, bei dem es sich um ein Typsynonym handelt, das den Typ bestehend aus einer Liste von Basisklassendeklarationen, einer Liste von Teamklassendeklarationen und einer Liste von Rollenklassendeklarationen zum Typ *prog* zusammenfasst. Alle in einem Programm enthaltenen Klassen müssen eindeutige Namen haben. Diese Einschränkung wird zum Nachweis verschiedener Hilfslemmata (siehe Abschnitt 5.5) benötigt und daher innerhalb des Wohlstrukturiertheitsprädikats für Programme ws_prog in Abschnitt 5.5 formalisiert. Durch die Definition des Prädikats ws_prog kann überprüft werden, ob ein Programm dieser Einschränkung genügt. Würde hingegen ein Typ mit einem entsprechenden Prädikat zur Formalisierung von Programmen definiert werden, könnten Programme, die dieser Einschränkung nicht genügen, erst gar nicht kodiert werden. Da dann keine Aussage über die Wohlstrukturiertheit von Programmen getroffen werden könnte, wird die Eindeutigkeit aller Klassennamen in einem Programm innerhalb der Definition des Prädikats ws_prog überprüft.

Klassendefinitionen sind als Elemente fixer Recordtypen dargestellt, da die Definitionen der Konstanten bclass, tclass, rclass und class auf dem eindeutigen Enthaltensein von Klassendeklarationen in der entsprechenden Klassendeklarationsliste basieren. Sie bilden die Basis für den Nachweis der Endlichkeit der in Abschnitt 5.4 vorgestellten Vererbungsrelationen. Die Zugriffe auf die Klassendeklarationslisten eines Programms werden durch die Konstanten bclass_decls, tclass_decls und rclass_decls modelliert.

types $prog = (bclass\_decl\ list * tclass\_decl\ list * rclass\_decl\ list)$

consts
  bclass_decls :: $prog \Rightarrow (tname * class)list$
  tclass_decls :: $prog \Rightarrow (tname * tclass)list$
  rclass_decls :: $prog \Rightarrow (tname * rclass)list$

defs
  bclass_decls_def:
  bclass_decls G == fst G

tclass_decls_def:
tclass_decls G == fst (snd G)

rclass_decls_def:
rclass_decls G == snd (snd G)

consts
bclass :: $prog \Rightarrow (tname, class)table$
tclass :: $prog \Rightarrow (tname, tclass)table$
rclass :: $prog \Rightarrow (tname, rclass)table$
class :: $prog \Rightarrow (tname, class)table$

defs
bclass_def:
bclass G C == table_of (bclass_decls G) C

tclass_def:
tclass G C == table_of (tclass_decls G) C

rclass_def:
rclass G C == table_of (rclass_decls G) C

class_def:
class G C ==
table_of ((bclass_decls G)
(map (λ(C,c). (C, class.truncate c)) (tclass_decls G))
(map (λ(C,c). (C, class.truncate c)) (rclass_decls G))) C

Die Prädiakte is_bclass, is_tclass, is_rclass und is_class überprüfen, ob es sich bei dem Namen einer Klasse C um eine im Programm G sichtbare Klasse der entsprechenden Art handelt.

consts
is_bclass :: $prog \Rightarrow tname \Rightarrow bool$
is_rclass :: $prog \Rightarrow tname \Rightarrow bool$
is_tclass :: $prog \Rightarrow tname \Rightarrow bool$
is_class :: $prog \Rightarrow tname \Rightarrow bool$

defs
is_bclass_def:
is_bclass G C == bclass G C $\neq$ None

is_tclass_def:
is_tclass G C == tclass G C $\neq$ None

is_rclass_def:
is_rclass G C == rclass G C $\neq$ None

is_class_def:
is_class G C ==
(bclass G C $\neq$ None) | (tclass G C $\neq$ None) | (rclass G C $\neq$ None)

5.4 Modellierung von Vererbungsrelationen

Die Vererbungsbeziehungen zwischen Klassen eines ObjectTeams/Java Programms G sind durch die Relationen sub_class1 G und sub_vclass1 G modelliert.

Die Definition der Relation sub_class1 G dient zur Formalisierung der `extends`-Vererbungsbeziehung zwischen Klassen und stellt die Basis für die Modellierung einer Subtypbeziehung zwischen Typen in Abschnitt 6.2 dar. Einschränkungen hinsichtlich der Vererbungsbeziehungen zwischen Klassen unterschiedlicher Klassenarten entlang der `extends`-Vererbungshierarchie sind innerhalb der Wohlstrukturiertheitsprädikate in Abschnitt 5.5 definiert.

Die durch die Relation sub_vclass1 G formalisierte implizite Vererbungsbeziehung zwischen Rollenklassen, die in voneinander erbenden Teamklassen enthalten sind, etabliert keine Subtypbeziehung zwischen durch diese Rollenklassen gebildeten Typen [Her03c]. Das bedeutet, dass Typen, die durch Rollenklas-

sen unterschiedlicher Teamklassen gebildet werden, nicht zueinander konvertierbar sind. Weitere Einschränkungen hinsichtlich der Konvertierbarkeit zwischen Typen werden in Abschnitt 6.2 definiert.

Die Vererbungsrelationen zwischen Klassen in ObjectTeams/Java werden als induktiv definierte Mengen durch Formulierung von *introduction rules* [NPW02] modelliert, die zum Ausdruck bringen, welchen Einschränkungen Elemente dieser Mengen genügen müssen. Ein Element ist durch Entsprechung einer *introduction rule* Bestandteil einer induktiv definierten Menge. Bei der Definition von induktiv definierten Mengen generiert Isabelle/HOL unter anderem Induktionsregeln und Regeln für Fallunterscheidungen [NPW02]. Diese werden zum Beispiel beim Nachweis von Aussagen über die ebenfalls mittels induktiver Mengendefinitionen formalisierten Typisierungs- und Auswertungsregeln in Abschnitt 6.3.4 beziehungsweise 8.2 verwendet.

```
consts
  sub_class1 :: prog ⇒ (tname * tname)set
  sub_vclass1 :: prog ⇒ (tname * tname)set
```

$$\frac{\mathsf{C} \neq \mathsf{Object} \qquad \mathsf{class\ G\ C} = \mathsf{Some\ c}}{(\mathsf{C}, \mathsf{super\ c}) \in (\mathsf{sub\_class1\ G})}$$

$$\frac{\mathsf{rclass\ G\ C} = \mathsf{Some\ c} \qquad \mathsf{implicit\ c} = \mathsf{Some\ is}}{(\mathsf{C}, \mathsf{the\ (implicit\ c)}) \in (\mathsf{sub\_vclass1\ G})}$$

Anstatt die Vererbungsrelationen als induktiv definierte Mengen zu formalisieren, können sie auch durch Konstantendefinitionen definiert werden (siehe sub_class1def und sub_vclass1def). Die Gleichheit der jeweiligen Vererbungsrelationen ist durch Nachweis der Lemmata sub_class1_sub_class1def_eq und sub_vclass1_sub_vclass1def_eq gezeigt.

Innerhalb der Definition der Konstanten sub_class1def wird die syntaktische Abkürzung c ← class G C. super c = D verwendet, die zum Ausdruck bringt, dass dem Klassennamen C im Programm G eine Klassendefinition c zugeordnet ist, die die Formel super c = D erfüllt.

```
consts
  sub_class1def :: prog ⇒ (tname * tname)set
  sub_vclass1def :: prog ⇒ (tname * tname)set

defs
sub_class1def_def:
  sub_class1def G ==
    {(C,D). C ≠ Object & (c ← class G C. super c = D)}

sub_vclass1def_def:
  sub_vclass1def G ==
    {(C,D). (c ← rclass G C. implicit c ≠ None & the (implicit c) = D)}

lemma sub_class1_sub_class1def_eq:
  sub_class1 G = sub_class1def G

lemma sub_vclass1_sub_vclass1def_eq:
  sub_vclass1 G = sub_vclass1def G
```

5.5 Wohlstrukturiertheit von Programmen

Es werden Prädikate zur Modellierung der Wohlstrukturiertheit von Klassendeklarationen und Programmen modelliert. Sie dienen erstens zur Überprüfung von für Object Teams spezifischen Einschränkungen hinsichtlich der erlaubten `extends`-Vererbungsbeziehungen zwischen Klassen. Zweitens werden Einschränkungen modelliert, die durch die Modellierung eines flachen Typnamensraumes explizit überprüft werden müssen. Drittens wird überprüft, ob es Zyklen in den Vererbungshierarchien gibt. Diese Überprüfung ist im Hinblick auf den Nachweis der Wohlfundiertheit der Vererbungsrelationen in Abschnitt 5.5 nötig, weil dabei unter anderem gezeigt werden muss, dass die Vererbungsrelationen azyklisch sind.

Die Einschränkungen hinsichtlich der erlaubten `extends`-Vererbungsbeziehungen zwischen Klassen und die Überprüfungen aufgrund der Modellierung eines flachen Typnamensraumes sind innerhalb der Prädikate is_ws_bclass, is_ws_tclass und is_ws_rclass definiert.

```
consts
  is_ws_bclass :: prog ⇒ tname ⇒ bool
  is_ws_tclass :: prog ⇒ tname ⇒ bool
  is_ws_rclass :: prog ⇒ tname ⇒ bool
```

Eine Basisklasse mit dem Namen C ist wohlstrukturiert, wenn im Programm G diesem Klassennamen eine Basisklassendefinition c zugeordnet ist, und, falls der Klassenname nicht Object ist, ihre Superklasse eine Basisklasse darstellt.

Eine Basisklasse darf nicht von einer Rollenklasse erben, weil es sonst zu Konflikten bei der Wiederverwendung von geerbten Methoden kommen würde, die eine Referenz auf ein umschließendes Objekt erwarten - Basisobjekte stellen aber keine in anderen Objekten enthaltenen Instanzen dar.

```
defs
  is_ws_bclass_def:
  is_ws_bclass G C == (c ← bclass G C. C ≠ Object ⟶ bclass G (super c) ≠ None)
```

Es wird vereinfachend davon ausgegangen, dass bei der Bildung einer Subteamklasse alle Rollenklassen der Superteamklasse redefiniert werden. Dadurch kann in den folgenden Definitionen die Eigenschaft verwendet werden, dass einer impliziten Superrollenklasse einer Superteamklasse immer genau eine Rollenklasse in einer Subteamklasse zugeordnet ist, die diese implizite Superrollenklasse redefiniert. Es wird davon ausgegangen, dass innerhalb eines Vorverarbeitungsprozesses für alle Rollenklassen der Superteamklasse, die nicht explizit in der Subteamklasse redefiniert werden, eine leere Rollenklassendeklaration angelegt wird, die in der Subteamklasse enthalten ist und mit der konzeptionell gleichnamigen Rollenklasse der Superteamklasse in einer impliziten Vererbungsbeziehung steht. Dies ist in Abbildung 4 und im Anhang bei der Kodierung des Ausschnitts des Brettspiel-Rahmenwerks veranschaulicht.

Eine Teamklasse mit dem Namen C ist wohlstrukturiert, wenn im Programm G diesem Klassennamen eine Teamklassendefinition c zugeordnet ist, alle in der Teamklassendefinition enthaltenen Rollenklassen unterschiedliche Namen besitzen und, falls der Klassenname der Teamklasse nicht Team ist, folgendes gilt.

- Die Teamklasse besitzt eine im Programm G sichtbare Superteamklasse [Her03c] (§1.2).
- Für alle in der Teamklasse C enthaltenen Rollenklassen gilt, dass diese, wenn sie unterschiedliche Namen haben, nicht voneinander entlang der impliziten Vererbungshierarchie erben. Weiterhin gilt, dass wenn die Superteamklasse der Teamklasse C nicht die ObjectTeams/Java-Standardklasse namens Team darstellt, alle in dieser Superteamklasse enthaltenen Rollenklassendefinitionen wohlstrukturiert sind, die umschließende Teamklasse aller Rollenklassen der Superteamklasse die Superteamklasse darstellt, und jeder Rollenklasse der Superteamklasse genau eine Rollenklasse der Subteamklasse C zugeordnet ist, die als implizite Superrollenklasse die Rollenklasse der Superteamklasse besitzt.

Wenn die Klasse C die ObjectTeams/Java-Standardklasse namens Team darstellt, dann soll gelten, dass sie keine Rollenklassen enthält, und ihre Superklasse eine wohlstrukturierte Basisklasse darstellt. Diese muss die Java-Standardklasse namens Object darstellen.

```
defs
  is_ws_tclass_def:
  is_ws_tclass G C ==
    (c ← tclass G C.
      (distinct (roles c)) &
      (C ≠ Team ⟶
        ((tclass G (super c) ≠ None) &
        (∀ R2 ∈ set (roles c).
          (is_ws_rclass G R2) & (encl_class (the (rclass G R2)) = C) &
          (∀ R1 ∈ set (roles c).
            R1 ≠ R2 ⟶ (R2, R1) ∉ (sub_vclass1 G)⁺ &
              (R1, R2) ∉ (sub_vclass1 G)⁺)) &
        ((super c ≠ Team) ⟶
          (∀ R1 ∈ set(roles (the (tclass G (super c))))).
            ((is_ws_rclass G R1) & (encl_class (the (rclass G R1)) = super c) &
```

```
            (∃ R21 ∈ set(roles c).
              (the (implicit (the (rclass G R21))) = R1))))))) &
      (C=Team ⟶ ((roles c = []) & (is_ws_bclass G (super c)) & (super c = Object))))
```

Eine Rollenklasse mit dem Namen C ist wohlstrukturiert, wenn im Programm G diesem Klassennamen eine Rollenklassendefinition c zugeordnet ist, es sich bei ihrer umschließenden Teamklasse nicht um die ObjectTeams/Java-Standardklasse namens Team handelt und folgendes hinsichtlich der die Rollenklasse C umschließenden, im Programm G enthaltenen Teamklassendefinition t und der Superklasse der Rollenklasse C entlang der `extends`-Vererbungshierarchie gilt.

- Die Rollenklasse C ist in der Liste der Rollenklassen der sie umschließenden Teamklasse enthalten.
- Wenn die Rollenklasse entlang der `extends`-Vererbungshierachie nicht von einer im Programm G sichtbaren Rollenklasse erbt, dann erbt sie von einer im Programm G enthaltenen Basisklasse.

  Die Basisklasse, von der eine Rollenklasse erbt, wird im folgenden als Superbasisklasse einer Rollenklasse bezeichnet.
- Wenn die Rollenklasse entlang der `extends`-Vererbungshierachie von einer im Programm G sichtbaren Rollenklasse erbt, dann muss diese Superrollenklasse in derselben Teamklasse enthalten sein wie die Rollenklasse C.

  Eine Rollenklasse darf nicht von einer in einer anderen Teamklasse enthaltenen Rollenklasse entlang der `extends`-Vererbungshierarchie erben, weil es sonst zu Konflikten hinsichtlich des Kontextes, in dem Instanzen der Subrollenklasse enthalten sind, kommen würde.

Falls die Rollenklasse C entlang der impliziten Vererbungshierarchie von einer Rollenklasse erbt, muss folgendes gelten.

- Die implizite Superrollenklasse der Rollenklasse C ist im Programm G sichtbar.
- Die die implizite Superrollenklasse der Rollenklasse C umschließende Teamklasse ist im Programm G sichtbar, es handelt sich bei ihr nicht um die ObjectTeams/Java-Standardklasse namens Team, und sie stellt die Superklasse der die Rollenklasse C umschließenden Teamklasse dar. Weiterhin gilt, dass die Superklasse einer Rollenklasse entlang der impliziten Vererbungshierarchie unverändert bleibt, und die implizite Superrollenklasse der Rollenklasse C in der Liste der Rollenklassen der sie umschließenden Teamklasse enthalten ist.

```
defs
  is_ws_rclass_def:
  is_ws_rclass G C ==
    (c ← rclass G C.
      (encl_class c ≠ Team) &
      (t ← tclass G (encl_class c). C ∈ set (roles t)) &
      (rclass G (super c) = None ⟶ bclass G (super c) ≠ None) &
      (rclass G (super c) ≠ None ⟶
        ((encl_class c = encl_class (the (rclass G (super c)))) &
        super c ∈ set(roles (the (tclass G (encl_class c)))))) &
      (implicit c ≠ None ⟶
        (vc ← rclass G (the (implicit c)).
          (tclass G (encl_class vc) ≠ None) & ((encl_class vc) ≠ Team) &
          (super (the (tclass G (encl_class c))) = encl_class vc) &
          (super c = super vc) &
          (the (implicit c) ∈ set (roles (the (tclass G (encl_class vc))))))))
```

Die Wohlstrukturiertheitsprädikate ws_b, ws_t, ws_r und ws_v überprüfen, ob die Vererbungsrelationen sub_class1 G und sub_vclass1 G azyklisch sind. Dies wird für den Nachweis der Wohlfundiertheit der Vererbungsrelationen benötigt.

```
consts
    ws_b :: prog ⇒ tname ⇒ tname ⇒ bool
    ws_t :: prog ⇒ tname ⇒ tname ⇒ bool
```

ws_r :: *prog* ⇒ *tname* ⇒ *tname* ⇒ *bool*
ws_v :: *prog* ⇒ *tname* ⇒ *tname* ⇒ *bool*

`defs`
ws_b_def:
ws_b G C D == (C ≠ Object ⟶ (is_ws_bclass G D & (D,C) ∉ (sub_class1 G)$^+$))

ws_t_def:
ws_t G C D == (C ≠ Team ⟶ (is_ws_tclass G D & (D,C) ∉ (sub_class1 G)$^+$))

ws_r_def:
ws_r G C D == (is_ws_bclass G D | is_ws_rclass G D) & (D,C) ∉ (sub_class1 G)$^+$

ws_v_def:
ws_v G C D == is_ws_rclass G D & (D,C) ∉ (sub_vclass1 G)$^+$

`lemma` ws_vD:
[|rclass G C = Some c; is_rclass G C; implicit c ≠ None; ws_prog G|]
⟹ is_ws_rclass G (the (implicit c)) & (the (implicit c),C) ∉ (sub_vclass1 G)$^+$

`lemma` ws_cD:
[|class G C = Some c; C ≠ Object; ws_prog G|]
⟹ is_class G (super c) & (super c, C) ∉ (sub_class1 G)$^+$

Das Prädikat ws_prog modelliert die Wohlstrukturiertheit von ObjectTeams/Java-Programmen. Ein Programm G ist wohlstrukturiert, wenn folgendes gilt.

- Die Java-Standardklasse namens Object ist in der Liste der Basisklassendeklarationen enthalten.
- Die Java-Standard-Exception-Klassen sind in der Liste der Basisklassendeklarationen enthalten.
- Die ObjectTeams/Java-Standardklasse namens Team ist in der Liste der Teamklassendeklarationen enthalten.
- Alle in der Basisklassendeklarationsliste des Programms G enthaltenen Basisklassendeklarationen sind für sich alleine und in Kombination mit ihrer Superklasse wohlstrukturiert.
- Alle in der Teamklassendeklarationsliste des Programms G enthaltenen Teamklassendeklarationen sind für sich alleine und in Kombination mit ihrer Superklasse wohlstrukturiert.
- Alle in der Rollenklassendeklarationsliste des Programms G enthaltenen Rollenklassendeklarationen sind für sich alleine und in Kombination mit ihrer Superklasse entlang der `extends`-Vererbungshierarchie wohlstrukturiert.
- Alle in der Rollenklassendeklarationsliste des Programms G enthaltenen Rollenklassendeklarationen sind, wenn sie eine implizite Superrollenklasse besitzen, in Kombination mit ihrer impliziten Superrollenklasse wohlstrukturiert.
- Alle in einem Programm enthaltenen Klassendeklarationen haben eindeutige Namen. Diese Einschränkung wird unter anderem zum Nachweis der Lemmata distinct_b_tr, distinct_t_br und distinct_r_bt benötigt, die an verschiedenen Stellen der Modellierung bei Fallunterscheidungen hinsichtlich der Art einer Klasse verwendet werden, und unter anderem zum Nachweis der Lemmata class_is_class und is_class_the_class, die in Abschnitt 5.6 beim Nachweis, dass die `extends`-Vererbungshierarchie endlich ist, verwendet werden.

`consts`
ws_prog :: *prog* ⇒ *bool*

`defs`
ws_prog_def:
ws_prog G ==
(Object,ObjectC) ∈ set (bclass_decls G) &
(SXcpt NullPointer, NullPointerC) ∈ set (bclass_decls G) &
(SXcpt OutOfMemory, OutOfMemoryC) ∈ set (bclass_decls G) &

```
    (SXcpt ClassCast,ClassCastC) ∈ set (bclass_decls G) &
    (Team,TeamC) ∈ set (tclass_decls G) &
    (∀ (C,c) ∈ set (bclass_decls G). is_ws_bclass G C & ws_b G C (super c)) &
    (∀ (C,c) ∈ set (tclass_decls G). is_ws_tclass G C & ws_t G C (super c)) &
    (∀ (C,c) ∈ set (rclass_decls G). is_ws_rclass G C & ws_r G C (super c)) &
    (∀ (C,c) ∈ set (rclass_decls G). ((implicit c ≠ None) ⟶ ws_v G C (the (implicit c)))) &
  (unique ((bclass_decls G)
    (map (λ(C,c). (C, class.truncate c)) (tclass_decls G))
      (map (λ(C,c). (C, class.truncate c)) (rclass_decls G))))
```

```
lemma distinct_b_tr:
  [|ws_prog G; is_bclass G C|]
  ⟹ (¬(is_tclass G C) & ¬(is_rclass G C))
```

```
lemma distinct_t_br:
  [|ws_prog G; is_tclass G C|]
  ⟹ (¬(is_bclass G C) & ¬(is_rclass G C))
```

```
lemma distinct_r_bt:
  [|ws_prog G; is_rclass G C|]
  ⟹ (¬(is_bclass G C) & ¬(is_tclass G C))
```

```
lemma is_class_the_class:
  [|ws_prog G; is_class G C|]
  ⟹ class G C = Some (the (class G C))
```

```
lemma class_is_class:
  [|ws_prog G; class G C = Some c|]
  ⟹ is_class G C
```

5.6 Nachweis der Wohlfundiertheit der Vererbungsrelationen

Für den Nachweis der Wohlfundiertheit der `extends`- und der impliziten Vererbungsrelation sub_class1 G und sub_vclass1 G ist zu zeigen, dass die Relationen sub_class1 G und sub_vclass1 G endlich und azyklisch sind. Der Aufbau dieser Nachweise ist aus [KNO+02b] übernommen.

Der Nachweis der Endlichkeit der `extends`-Vererbungsrelation sub_class1 G besteht aus folgenden Schritten.

- Unter der Voraussetzung, dass alle in einem Programm enthaltenen Namen von Klassendeklarationen eindeutig sind, wird die innerhalb des Lemmas sub_class1_def2 formulierte Gleichheit der beiden Mengen nachgewiesen.
- Die Endlichkeit der mit der Konstanten Sigma definierten Menge wird durch Nachweis der Endlichkeit der Menge {C. is_class G C} und durch Nachweis der Endlichkeit der Menge {D. C ≠ Object & super (the (class G C)) = D} für ein beliebiges Element der Menge {C. is_class G C}, von dem die erste Menge abhängt, gezeigt.

```
lemma sub_class1_def2:
  ws_prog G
  ⟹ sub_class1 G = (SIGMA C : {C. is_class G C} . {D. C ≠ Object & super (the (class G C)) = D})
```

```
lemma finite_sub_class1:
  ws_prog G ⟹ finite (sub_class1 G)
```

```
lemma finite_is_class:
  ws_prog G ⟹ finite {C. is_class G C}
```

Das Prädikat finite ist als syntaktische Abkürzung in [Pau+02b] (Finite_Set.thy) definiert und bringt zum Ausdruck, dass eine Menge Element der induktiv definierten Menge Finites aller endlicher Mengen ist.

Das Lemma finite_is_class besagt, dass die Menge aller in einem wohlstrukturierten Programm G enthaltenen Namen von Klassendeklarationen endlich ist. Dies wird unter anderem durch Anwendung des Lemmas finite_dom_map_of aus [Pau+02b] (Map.thy) gezeigt. Das Lemma finite_dom_map_of besagt, dass

die Domäne einer mit Hilfe der Konstanten map_of in einen *table* konvertierten Liste, die per Definition endlich ist, ebenfalls endlich ist. (table_of ist als syntaktische Abkürzung für die Konstante map_of definiert.)

Dass die `extends`-Vererbungsrelation azyklisch ist, wird mit Hilfe des Lemmas sub_class1_irrefl_lemma1 gezeigt, für dessen Nachweis das Lemma ws_cD (siehe Abschnitt 5.5) verwendet wird.

```
lemma sub_class1_irrefl_lemma1:
  ws_prog G ⟹ (sub_class1 G)^-1 Int (sub_class1 G)^+ = {}

lemma acyclic_sub_class1:
  ws_prog G ⟹ acyclic (sub_class1 G)
```

Unter der Voraussetzung, dass ein Programm G wohlstrukturiert ist, wird die Wohlfundiertheit der Vererbungsrelation sub_class1 G und in Anlehnung an [KNO+02b] die Wohlfundiertheit der inversen Vererbungsrelation (sub_class1 G)^-1 durch Anwendung der Lemmata finite_sub_class1 und acyclic_sub_class1 gezeigt.

```
lemma wellfounded_sub_class1_:
  ws_prog G ⟹ wf (sub_class1 G)

lemma wellfounded_sub_class1:
  ws_prog G ⟹ wf ((sub_class1 G)^-1)
```

Entsprechend dem Nachweis der Wohlfundiertheit der `extends`-Vererbungsrelation sub_class1 G wird die Wohlfundiertheit der impliziten Vererbungsrelation sub_vclass1 G gezeigt.

```
lemma sub_vclass1_def2:
  sub_vclass1 G =
    (SIGMA C : {C. is_rclass G C} .
      {D. implicit (the (rclass G C)) ≠ None & the (implicit (the (rclass G C))) = D})

lemma finite_is_rclass:
  finite {C. is_rclass G C}

lemma finite_sub_vclass1:
  finite (sub_vclass1 G)

lemma sub_vclass1_irrefl_lemma1:
  ws_prog G ⟹ (sub_vclass1 G)^-1 Int (sub_vclass1 G)^+ = {}

lemma acyclic_sub_vclass1:
  ws_prog G ⟹ acyclic (sub_vclass1 G)

lemma wellfounded_sub_vclass1_:
  ws_prog G ⟹ wf (sub_vclass1 G)

lemma wellfounded_sub_vclass1:
  ws_prog G ⟹ wf ((sub_vclass1 G)^-1)
```

5.7 Formulierung von Induktionsschemata über die Vererbungsrelationen

Es werden Induktionsschemata über die `extends`- und die implizite Vererbungsrelation gebaut, die Instanzen der Induktionsregel für *well-founded* Relationen [NPW02] darstellen. Die Induktionsschemata sub_class1_induct und sub_vclass1_induct bringen folgendes zum Ausdruck. Um zu zeigen, dass eine Eigenschaft P für eine bestimmte Klasse A gilt, reicht es zu zeigen, dass die Eigenschaft P für eine beliebige Klasse C gilt unter der Voraussetzung, dass die Eigenschaft P für die Superklasse dieser Klasse entlang der `extends`- beziehungsweise der impliziten Vererbungshierarchie gilt. Außerdem muss das Programm G, relativ zu dem die `extends`- beziehungsweise die implizite Vererbungshierarchie gebildet sind, wohlstrukturiert sein.

```
lemma sub_class1_induct:
  [|ws_prog G;
  !!C. ∀ c. (C,super c) ∈ (sub_class1 G) ⟶ P (super c) ⟹ P C |]
  ⟹ P A
```

```
lemma sub_vclass1_induct:
  [|ws_prog G;
  !!C. ∀ c. (C,the (implicit c)) ∈ (sub_vclass1 G) ⟶
                P (the (implicit c)) ⟹ P C |]
  ⟹ P A
```

Aus den Induktionsschemata sub_class1_induct und sub_vclass1_induct werden spezifischere Induktionsschemata ws_sub_class1_induct und ws_sub_vclass1_induct abgeleitet, die zum Beispiel in Abschnitt 7.2 zum Nachweis der statischen Schichtenaussage verwendet werden.

```
lemma ws_sub_class1_induct:
  [|is_class G C; ws_prog G;
  !! C c. [|class G C = Some c; C ≠ Object ⟶ (is_class G (super c) &
           (C, super c) ∈ (sub_class1 G) & P (super c))|] ⟹ P C|]
  ⟹ P C

lemma ws_sub_vclass1_induct:
  [|is_rclass G C; ws_prog G;
  !! C c. [|rclass G C = Some c; implicit c ≠ None ⟶
          (is_rclass G (the (implicit c)) &
           (C, the (implicit c)) ∈ (sub_vclass1 G) & P (the (implicit c)))|]
        ⟹ P C|]
  ⟹ P C
```

5.8 Modellierung von Traversierungsfunktionen zur Formalisierung des Zugriffs auf Eigenschaften von Klassen

Es werden Funktionen zur Traversierung von Klassenhierarchien, die auf der `extends`- und der impliziten Vererbungsrelation basieren, als total rekursive Funktionen modelliert.

Die Funktion class_rec ist über die wohlfundierte Vererbungsrelation sub_class1 G unter Verwendung des Kombinators same_fst definiert, da das erste Argument bei einem rekursiven Aufruf der Funktion class_rec - das Programm G - unverändert bleibt [NPW02]. Das Argument f der Funktion class_rec stellt eine Funktion dar, die den Namen der Klasse, die Definition dieser Klasse und das Argument t eines rekursiven Aufrufs der Funktion class_rec kombiniert. Angewendet wird die Funktion zum Beispiel zur Bestimmung aller Felder, auf die eine Klasse Zugriff hat, die sie also selbst definiert und erbt (siehe Abschnitt 6.3.3).

Entsprechend der Definition der Konstanten class_rec wird die Konstante vclass_rec zur Traversierung von Klassenhierarchien entlang der impliziten Vererbungshierarchie definiert.

```
consts
  class_rec :: prog * tname ⇒ 'a ⇒ (tname ⇒ class ⇒ 'a ⇒ 'a) ⇒ 'a
  vclass_rec :: prog * tname ⇒ 'a ⇒ (tname ⇒ rclass ⇒ 'a ⇒ 'a) ⇒ 'a

recdef class_rec same_fst ws_prog (λG. (sub_class1 G)^-1)
  class_rec (G,C) =
    (λt f. case class G C of
      None ⇒ arbitrary
      | Some c ⇒ if (ws_prog G) then f C c (if (C = Object) then t
                    else class_rec (G,super c) t f) else arbitrary)

recdef vclass_rec same_fst ws_prog (λG. (sub_vclass1 G)^-1)
  vclass_rec (G,C) =
    (λt f. case rclass G C of
      None ⇒ arbitrary
      | Some c ⇒ if (ws_prog G) then f C c (if (implicit c = None) then t
                    else vclass_rec (G,the (implicit c)) t f) else arbitrary)
```

6 Formalisierung eines Typsystems für ObjectTeams/Java

Es wird ein Typsystem für ObjectTeams/Java formalisiert. In Abschnitt 6.1 wird ein Modell instanzbasierter statischer und dynamischer Typen motiviert und aufgestellt. Beziehungen zwischen Typen werden in Abschnitt 6.2 modelliert. Sie sind mit dem Ziel definiert, *confined role objects* zu realisieren. *Confined role objects* stellen Rollenobjekte dar, die nur innerhalb der sie umschließenden Teaminstanz referenziert werden. In Abschnitt 9.2 wird eine Aussage über das *Confinement* von Rollenobjekten betrachtet. Regeln zur Typisierung von Termen relativ zu einem statischen Environment und die Modellierung der Bindung von Rollentypen abhängig vom Teamkontext des statischen Environments zur Realisierung der kovarianten Redefinition von Rollenklassen werden in Abschnitt 6.3 vorgestellt.

Die Modellierungen von Typrelationen und Typisierungsregeln orientieren sich hinsichtlich ihrer Definitionen als induktiv definierte Mengen an den in [KNO+02b] vorgestellten Formalisierungen zur Definition von Relationen zwischen Typen in Java und zur Definition von Regeln zur Typisierung von Java-Konstrukten.

6.1 Instanzbasierte Typen

6.1.1 Motivation für ein Modell instanzbasierter Typen

Bei der Verfeinerung einer Teamklasse werden alle in ihr enthaltenen Rollenklassen implizit geerbt und redefiniert. Das bedeutet, dass innerhalb eines Teamkontextes nur die Rollenklassen sichtbar sind, die in dem Teamkontext enthalten sind. Dies hat für die Modellierung eines Typsystems für ObjectTeams/Java zwei Konsequenzen.

Erstens soll das Modell von Typen in ObjectTeams/Java zum Ausdruck bringen, dass die Überprüfung von Rollentypen von dem Teamkontext abhängt, in dem sie auftreten.

Zweitens müssen in allgemeineren Teamkontexten deklarierte und denotierte Rollentypen in Attribut- und Methodendeklarationen abhängig von dem Teamkontext, in dem sie wiederverwendet werden, gebunden werden. Nur dadurch kann unter anderem sichergestellt werden, dass die kovariante Redefinition von Rollenklassen nicht zu Typfehlern führt. Denn diese hat eine Spezialisierung von durch Rollenklassen gebildeteten Methodenparametertypen beim Erben und Überschreiben von Methoden während der Verfeinerung von Teamkontexten zur Folge.

Während der statischen Analyse ist der Teamkontext, abhängig von dem statische Typen gebunden werden, der statische Typ der modellierungsspezifischen Variablen namens TContext (siehe Abschnitt 5.1.1), um die das Environment erweitert ist. (Die Typisierungsregeln für ObjectTeams/Java-Konstrukte sind relativ zu diesem Environment definiert und dadurch implizit mit dem Teamkontext parametrisiert.) Diese Variable enthält eine Referenz auf das aktuelle Teamobjekt. Zur Laufzeit kann sie ein spezielleres Teamobjekt referenzieren. Das bedeutet, dass der statische Typ dieser Variablen nur eine obere Schranke hinsichtlich des dynamischen Typs dieses Teamobjekts darstellt - das Teamobjekt kann zur Laufzeit ein Element eines beliebig spezielleren Subtyps des statischen Typs der das aktuelle Teamobjekt referenzierenden Variablen sein.

Um festzustellen, ob der dynamische Typ eines Rollenobjekts konform zum statischen durch eine Rollenklasse gebildeten Typ eines Terms ist, dessen Wert auf ein Rollenobjekt verweist, muss folgendes untersucht werden. Die die Rollenobjekte umschließenden Teaminstanzen müssen denselben dynamischen Typ haben. Dann ist erstens sichergestellt, dass beide Rollenobjekte den Anforderungen genügen, die im Kontext dieser Teaminstanzen gestellt werden. Diese Teaminstanzen müssen also nur denselben dynamischen Typ haben, sie müssen nicht dieselbe Teaminstanz darstellen. Zweitens können die tatsächlichen Typen der Rollenobjekte durch die Kenntnis der dynamischen Typen der Teaminstanzen, in denen sie enthalten sind, bestimmt werden. Infolgedessen kann überprüft werden, ob ihre Typen konform sind, also ob die Rollenobjekte Instanzen von voneinander entlang der `extends`-Vererbungshierarchie erbenden Rollenklassen oder Instanzen derselben Rollenklasse darstellen. Das bedeutet, dass für die Typsicherheit von ObjectTeams/Java die Kenntnis der dynamischen Typen der umschließenden Teaminstanzen zweier Rollenobjekte ausreicht, um festzustellen, ob der dynamische Typ eines Rollenobjekts konform zu dem statischen durch eine Rollenklasse gebildeten Typ eines Terms ist, dessen Wert auf ein Rollenobjekt verweist. Diese Forderung wird in ObjectTeams/Java verstärkt. Es wird nicht nur gefordert, dass die die Rollenobjekte umschließenden Teaminstanzen denselben dynamischen Typ haben, sondern dass sie dieselbe Instanz darstellen. In ObjectTeams/Java wird erstens der statische Typ der Variablen, die auf das aktuelle Teamobjekt verweist, verwendet, um die statischen Typen von Variablen, die Rollenobjekte referenzie-

ren, einzugrenzen. Zweitens wird die Substituierbarkeit von Rollenobjekten zwischen unterschiedlichen Teaminstanzen verboten. Statisch ist bekannt, dass gerade die Rollenobjekte, die in derselben Teaminstanz enthalten sind, den im Kontext dieser Teaminstanz gestellten Anforderungen genügen. Es wird also instanzbasiert überprüft, ob der dynamische Typ eines Rollenobjekts konform zum statischen durch eine Rollenklasse gebildeteten Typ eines Terms ist, dessen Wert auf ein Rollenobjekt verweist. Dies ist genau dann der Fall, wenn die beiden Rollenobjekte in derselben Teaminstanz enthalten sind und Instanzen derselben Rollenklasse oder Instanzen von voneinander entlang der `extends`-Vererbungshierarchie erbenden Rollenklassen darstellen. Das bedeutet, dass eine Teaminstanz nicht nur einen Container für Rollenobjekte darstellt, die zwischen Teaminstanzen mit demselben dynamischen Typ beliebig ausgetauscht werden können, sondern eine Menge von miteinander agierenden Rollenobjekten auf Objektebene kapselt.

Zur Formalisierung von Referenztypen in ObjectTeams/Java wird ein Modell instanzbasierter Referenztypen aufgestellt. Ein instanzbasierter Referenztyp stellt einen Typ dar, der neben der den Referenztyp bildenden Klasse um einen Instanzkontext angereichert ist, der zum Ausdruck bringt, relativ zu welcher Instanz der Typ verankert ist und gebunden wird. Das Modell instanzbasierter Typen wird in Abschnitt 6.1.3 vorgestellt. In Abbildung 4 sind alle in dem Ausschnitt des Brettspiel-Rahmenwerkes deklarierten Typen durch instanzbasierte Typen modelliert.

6.1.2 Entwurfsentscheidungen bei der Modellierung eines Typsystems für ObjectTeams/Java mit instanzbasierten Typen

Um statisch garantieren zu können, dass innerhalb eines Teamkontextes nur Rollenobjekte referenziert werden, die den Anforderungen des Teamkontextes genügen, muss folgendes überprüft werden.

Erstens muss sichergestellt werden, dass alle in einem Teamkontext auftretenden Rollentypen, deren Instanzkontext die in Team- und Rollenklassendeklarationen enthaltene Referenz auf das aktuelle beziehungsweise umschließende Teamobjekt darstellt, durch in diesem Teamkontext sichtbare Rollenklassen oder Superbasisklassen dieser Rollenklassen gebildet sind. Dies wird bei der Definition von Regeln zur Typisierung von ObjectTeams/Java-Konstrukten (siehe Abschnitt 6.3.4) und bei der Wohlgeformtheitsprüfung von Typdeklarationen (siehe Abschnitt 7.1.1) überprüft. Durch die in Abschnitt 7.2 formulierte statische Schichtenaussage wird nachgewiesen, dass das in dieser Arbeit modellierte Typsystem für ObjectTeams/Java diese Forderung erfüllt.

Zweitens enthält eine Konformitätsbeziehung zwischen instanzbasierten Typen sowohl eine Aussage über die Beziehung der die Typen bildenden Klassen, als auch über die Instanzkontexte der Typen. Zwei Typen sind zueinander konform, wenn die Klassen, durch die sie gebildet werden, voneinander entlang der `extends`-Vererbungsbeziehung erben, und ihre Instanzkontexte identisch sind (siehe Abschnitt 9.1). Auf Rollenobjekte bezogen bedeutet dies, dass nur Rollenobjekte substituiert werden können, deren Klassen gleich sind oder in einer Subklassenbeziehung stehen, und die in demselben Teamobjekt enthalten sind.

In dieser Arbeit werden nur Rollentypen betrachtet, deren Instanzkontexte die in Team- und Rollenklassendeklarationen enthaltene Referenz auf das aktuelle beziehungsweise umschließende Teamobjekt darstellen. Das bedeutet, dass die Sichtbarkeit von Rollenobjekten auf den Kontext der sie umschließenden Teaminstanz eingeschränkt ist. *Externalised roles* hingegen sind uneingeschränkt sichtbar (siehe Abschnitt 2.3). Ihre Typen sind um Instanzkontexte angereichert, die Variablen bezeichnen, die Referenzen auf die Teaminstanzen enthalten, in denen sie enthalten sind. Sie sollen durch die Modellierung zusätzlicher Einschränkungen dieselbe Typsicherheit von ObjectTeams/Java bieten wie nur innerhalb der sie umschließenden Teaminstanz sichtbare Rollenobjekte. Dabei muss überprüft werden, ob die Instanzkontexte, durch die die Rollentypen verankert sind, Referenzen auf dieselbe Teaminstanz enthalten.

Es wird davon ausgegangen, dass bei der Formalisierung von ObjectTeams/Java-Programmen alle Referenztypen um den Instanzkontext, von dem sie abhängen, explizit angereichert werden. Alle folgenden Modellierungen setzen eine Unterscheidbarkeit zwischen Rollentypen und Nicht-Rollentypen voraus. Nicht-Rollentypen stellen in dieser Arbeit Typen dar, die durch Team- oder Basisklassen gebildet sind, und die nicht relativ zu einer Instanz verankert sind. (Durch Schachtelung von Teamklassen können Teamklassen zu Rollen der äußeren Teamklasse werden. Dies wird im Rahmen dieser Arbeit nicht betrachtet.)

Als Rollentyp wird ein Typ aufgefasst, der durch die in Team- und Rollenklassendeklarationen enthaltene Referenz auf das aktuelle beziehungsweise umschließende Teamobjekt und eine Rollenklasse oder Superbasisklasse einer Rollenklasse gebildet ist. Die Modellierung von instanzbasierten Rollentypen bietet die Möglichkeit, die Weitungs- und Einengungsrelation zwischen Rollentypen durch die `extends`-Vererbungsbeziehung zwischen Rollenklassen zu definieren, und die Weitung eines durch eine Rollenklasse gebildeten Typs hin zu einem Typ, der durch eine Superbasisklasse dieser Rollenklasse gebildet ist, zu

erlauben. Dies könnte im Hinblick auf die Verwendung von rollentypspezifischen Container-Klassen interessant sein, um zum Beispiel eine Liste von Rollenobjekten mit einem gemeinsamen durch eine Basisklasse gebildeten Supertyp zu speichern. Dies wird innerhalb dieser Arbeit nicht realisiert.

6.1.3 Modellierung statischer instanzbasierter Typen

Die Modellierung primitiver Typen durch den Typ *prim_ty* ist aus [KNO+02a] übernommen. Der Konstruktor Void des Typs *prim_ty* formalisiert das Schlüsselwort `void` [GJS+00] (§8.4) zur Deklaration von Methoden ohne Rückgabewert.

`datatype` *prim_ty*
= Void
| Boolean
| Integer

Der Typ *ic* stellt den Instanzkontext dar, um den statische Referenztypen angereichert werden.

Der Konstruktor TheVoid wird zur Anreicherung statischer Referenztypen verwendet, die unabhängig von einer Instanz gebildet werden, also keine Rollentypen darstellen. Auf der Ebene von Objekten bedeutet dies, dass die auf einen Nicht-Rollentyp typisierten Variablen Objekte referenzieren, die nicht in anderen Objekten enthalten sind.

Im Gegensatz dazu soll der Konstruktor TThis zum Ausdruck bringen, dass durch diesen Instanzkontext gebildete Referenztypen abhängig von dem Teamkontext, in dem sie auftreten, gebildet werden. Dies bedeutet auf Objektebene, dass die auf Rollentypen typisierten Variablen Rollenobjekte referenzieren, die in Teamobjekten enthalten sind.

Der Konstruktor TThis des Typs *ic* verankert Typen relativ zu einer Instanz. Demgegenüber stellt Tthis einen Ausdruck vom Typ *expr* dar (siehe Abschnitt 5.2). Der Ausdruck Tthis modelliert die in Rollenklassendeklarationen enthaltene Referenz auf das aktuelle umschließende Teamobjekt. Die in Abschnitt 6.3.4 beziehungsweise 8.2 definierten Typisierungs- und Auswertungsregeln sind hingegen mit der modellierungsspezifischen Variablen TContext (siehe Abschnitt 5.1.1) parametrisiert. Diese stellt den Teamkontext dar, abhängig von dem Rollentypen gebunden werden.

In der konkreten Implementierung von ObjectTeams/Java werden alle innerhalb von Team- und Rollenklassendeklarationen enthaltenen, nicht qualifizierten Rollentypen wie zum Beispiel der Typ `Board` des Attributs `board` in Abbildung 1 bezogen auf dieses Beispiel interpretiert als `BoardGame.this.Board`. Bei der Formalisierung von ObjectTeams/Java wird diese implizite Verankerung von Rollentypen durch Anreicherung von Rollentypen um den Instanzkontext TThis explizit modelliert. Die Unabhängigkeit von Nicht-Rollentypen von einem Instanzkontext wird durch Anreicherung dieser Typen um den Instanzkontext TheVoid zum Ausdruck gebracht (siehe Abbildung 4).

`datatype` *ic*
= TheVoid
| TThis

Der Typ *ref_ty* modelliert Referenztypen. Der Konstruktor NullT dient zur Modellierung des Null-Typs, dem Typ des Ausdrucks `null`. Dieser Typ darf nicht in Variablendeklaration oder Cast-Ausdrücken vorkommen [GJS+00] (§4.1).

Statische instanzbasierte Referenztypen werden durch den Konstruktor ClassT modelliert. Dieser erhält ein Argument vom Typ eines Typnamens *tname* und ein Argument vom Typ eines statischen Instanzkontextes *ic*.

`datatype` *ref_ty*
= NullT
| ClassT *ic tname*

Primitive Typen vom Typ *prim_ty* und instanzbasierte Referenztypen vom Typ *ref_ty* werden durch den Typ *ty* zusammengefasst.

`datatype` *ty*
= PrimT *prim_ty*
| RefT *ref_ty*

Die syntaktischen Abkürzungen NT und Class werden zur besseren Lesbarkeit definiert.

```
syntax
    NT :: ty
    Class :: ic ⇒ tname ⇒ ty

translations
    NT == RefT NullT
    Class i C == RefT (ClassT i C)
```

Die über die Typen *ref_ty* und *ty* mittels primitiver Rekursion definierten Konstanten cls' und cls ermöglichen den Zugriff auf den Namen der Klasse, die einen Typ bildet. Analog dazu dienen die Konstanten ic' und ic zur Modellierung des Zugriffs auf den Instanzkontext, um den ein Typ angereichert ist.

```
consts
    cls' :: ref_ty ⇒ tname option
    cls :: ty ⇒ tname option

primrec
    cls' (NullT) = None
    cls' (ClassT i C) = Some C

primrec
    cls (PrimT pt) = None
    cls (RefT rt) = cls' rt

consts
    ic' :: ref_ty ⇒ ic option
    ic :: ty ⇒ ic option

primrec
    ic' (NullT) = None
    ic' (ClassT i C) = Some i

primrec
    ic (PrimT pt) = None
    ic (RefT rt) = ic' rt
```

6.1.4 Modellierung dynamischer instanzbasierter Typen

Während der statischen Analyse stellt der Instanzkontext von Referenztypen eine syntaktische Konstante dar. Jeder in einer Attribut- oder Methodendeklaration enthaltene Rollentyp ist um den Instanzkontext TThis angereichert. Zur Laufzeit kann diese Konstante auf die tatsächliche Referenz eines Rollenobjekts auf das Teamobjekt, in dem es enthalten ist, abgebildet werden. Die Semantik des Instanzkontextes von Rollentypen kann durch diese unveränderliche Referenz, die ein Rollenobjekt auf sein umschließendes Teamobjekt hat, definiert werden. Dadurch wird die Verschiedenheit von dynamischen Typen von Referenzen auf Rollenobjekte derselben Rollenklasse, die in unterschiedlichen Teaminstanzen enthalten sind, zum Ausdruck gebracht. Dies motiviert die explizite Modellierung dynamischer Typen durch den Typ *dyn_ty*. Ein dynamischer Typ einer Referenz auf ein Objekt besteht also neben der Klasse, dessen Instanz es ist, aus einem semantisch definierten Instanzkontext. Dieser ist durch den Typ *dyn_ic* formalisiert.

```
datatype dyn_ic
    = DynTheVoid
    | DynTId loc
```

Der Konstruktor DynTId erhält ein Argument vom Typ *loc*. Der Typ *loc* modelliert Referenzen auf Objekte. Der Konstruktor XcptRef formalisiert ausgezeichnete Referenzen auf Exception-Objekte. Der Konstruktor Loc modelliert Referenzen auf alle anderen Objekte. Er erhält ein Argument vom Typ *loc_*, der abstrakte Referenzen auf Objekte modelliert. Die Formalisierung von Referenzen auf Objekte ist aus [KNO+02a] übernommen.

```
typedecl loc_

datatype loc
    = XcptRef xcpt
```

```
  | Loc loc_

datatype dyn_ref_ty
  = DynNullT
  | DynClassT dyn_ic tname

datatype dyn_ty
  = DynPrimT prim_ty
  | DynRefT dyn_ref_ty
```

Analog zur Definition der syntaktischen Abkürzungen NT und Class werden die syntaktischen Abkürzungen DynNT und DynClass zur besseren Lesbarkeit definiert. Entsprechend dem Zugriff auf die einen statischen instanzbasierten Typ bildende Klasse und den Instanzkontext, um den dieser Typ angereichert ist, werden die Konstanten dyn_cls', dyn_cls, dyn_ic' und dyn_ic definiert.

```
syntax
  DynNT :: dyn_ty
  DynClass :: dyn_ic ⇒ tname ⇒ dyn_ty

translations
  DynNT == DynRefT DynNullT
  DynClass i C == DynRefT (DynClassT i C)

consts
  dyn_cls' :: dyn_ref_ty ⇒ tname option
  dyn_cls :: dyn_ty ⇒ tname option

primrec
  dyn_cls' (DynNullT) = None
  dyn_cls' (DynClassT i C) = Some C

primrec
  dyn_cls (DynPrimT pt) = None
  dyn_cls (DynRefT rt) = dyn_cls' rt

consts
  dyn_ic' :: dyn_ref_ty ⇒ dyn_ic option
  dyn_ic :: dyn_ty ⇒ dyn_ic option

primrec
  dyn_ic' (DynNullT) = None
  dyn_ic' (DynClassT i C) = Some i

primrec
  dyn_ic (DynPrimT pt) = None
  dyn_ic (DynRefT rt) = dyn_ic' rt
```

Die über den Typ *dyn_ic* mittels primitiver Rekursion definierte Konstante tid modelliert den Zugriff auf die Identität eines Teamobjekts, die den dynamischen Instanzkontext eines Rollentyps bildet.

```
consts
  tid :: dyn_ic ⇒ loc option

primrec
  tid (DynTheVoid) = None
  tid (DynTId l) = Some l
```

6.2 Modellierung statischer und dynamischer Typrelationen

Es werden eine syntaktische Subtyprelation (widen G), eine Relation zum Einengen von Referenztypen (narrow G) und eine Cast-Relation (cast G) zur Konvertierung von Typen modelliert. Sie werden als induktiv definierte Mengen formalisiert.

```
consts
  widen :: prog ⇒ (ty * ty)set
```

narrow :: $prog \Rightarrow (ty * ty)set$
cast :: $prog \Rightarrow (ty * ty)set$

6.2.1 Weitungsrelation zwischen statischen instanzbasierten Typen

Die syntaktische Subtyprelation zwischen statischen instanzbasierten Typen basiert auf der `extends`-Vererbungsbeziehung zwischen Klassen, die durch die Relation sub_class1 G formalisiert ist (siehe Abschnitt 5.4). Die zueinander weitbaren Typen werden durch die folgenden *introduction rules* als Elemente der Menge widen G beschrieben.

Die Konversion eines Typs zu sich selbst ist für jeden Typ erlaubt und wird als *identity conversion* bezeichnet [GJS+00] (§6.1.1).

$$\frac{}{(T, T) : \text{widen G}}$$

Der Null-Typ kann zu jedem beliebigen Referenztyp geweitet werden [GJS+00] (§4.1).

$$\frac{}{(NT, RefTR) \in \text{widen G}}$$

Nicht-Rollentypen können zu Nicht-Rollentypen entlang der `extends`-Vererbungshierarchie geweitet werden.

$$\frac{(C,D) \in (\text{sub\_class1 G})* \qquad \text{is\_bclass G C} \mid \text{is\_tclass G C} \qquad i = \text{TheVoid}}{\begin{array}{c}\text{is\_bclass G C} \longrightarrow \text{is\_bclass G D} \\ \text{is\_tclass G C} \longrightarrow (\text{is\_tclass G D} \mid (\text{is\_bclass G D} \;\&\; D = \text{Object}))\end{array}}{(\text{Class i C}, \text{Class i D}) \in (\text{widen G})}$$

Rollentypen können entlang der `extends`-Vererbungshierarchie geweitet werden. Entscheidend für die Realisierung von *confined role objects* (siehe Abschnitt 9.3) ist das Erhaltenbleiben des Instanzkontextes TThis bei der Weitung von Rollentypen. Dadurch wird sichergestellt, dass Rollenobjekte nicht durch Variablen referenziert werden können, die auf einen Nicht-Rollentyp typisiert sind.

In dieser Arbeit sind Typrelationen zur Realisierung von *confined role objects* modelliert. Opake Rollenobjekte stellen Rollenobjekte dar, die außerhalb des Teamkontextes, in dem sie enthalten sind, als Elemente eines Nicht-Rollentyps sichtbar sind. Für die Realisierung von opaken Rollenobjekten wäre die Definition einer weiteren *introduction rule* der syntaktischen Subtyprelation nötig. Diese müsste die Weitung von Rollentypen zu Nicht-Rollentypen entlang der `extends`-Vererbungshierarchie und die Weitung des Instanzkontextes von TThis nach TheVoid zulassen.

$$\frac{(C,D) \in (\text{sub\_class1 G})* \qquad \text{is\_rclass G C} \mid \text{is\_bclass G C} \qquad i = \text{TThis}}{\begin{array}{c}\text{is\_rclass G C} \longrightarrow (\text{is\_rclass G D} \mid \\ \text{is\_bclass G D} \;\&\; (\exists\, B.\ (B,D) \in (\text{sub\_class1 G})^+ \;\&\; \text{is\_rclass G B}) \\ \text{is\_bclass G C} \longrightarrow \text{is\_bclass G D} \;\&\; (\exists\, B.\ (B, D) \in (\text{sub\_class1G})^+ \;\&\; \text{is\_rclass G B})\end{array}}{(\text{Class i C}, \text{Class i D}) \in (\text{widen G})}$$

Die Transitivität der syntaktischen Subtypbeziehung zwischen statischen instanzbasierten Typen wird durch den Nachweis des Lemmas widen_trans per *rule induction* über die Menge widen G gezeigt. Eine Voraussetzung für diesen Nachweis ist, dass das Programm G, relativ zu dem die Subtyprelation definiert ist, wohlstrukturiert ist. Dies ist notwendig, um innerhalb des Subziels dieses Beweises bei der Betrachtung der Weitbarkeit von Teamtypen bis hin zum Typ Class TheVoid Object die Lemmata ws_prog_Object_is_bclass und sub_class_Object_eqD anwenden zu können.

`lemma` sub_class_Object_eqD:
(Object,D) ∈ (sub_class1 G)*⟹ D = Object

`lemma` ws_prog_Object_is_bclass:
ws_prog G ⟹ is_bclass G Object

`lemma` widen_trans [rule_format]:
[|(S,U) ∈ (widen G); ws_prog G|]
⟹ ∀ T. (U,T) ∈ (widen G) ⟶ (S,T) ∈ (widen G)

Das Prädikat widens modelliert die Überprüfung, ob die Typen zweier Listen statischer instanzbasierter Typen paarweise zueinander weitbar sind.

```
constdefs
  widens :: prog ⇒ ty list ⇒ ty list ⇒ bool
  widens G Ts Ts' == list_all2 (λT T'. (T,T') ∈ (widen G)) Ts Ts'
```

6.2.2 Einengungsrelation zwischen statischen instanzbasierten Typen

Die auf der `extends`-Vererbungshierarchie zwischen Klassen eines Programms G basierende Einengungsrelation narrow G zwischen statischen instanzbasierten Referenztypen ist analog zur Weitungsrelation zwischen statischen instanzbasierten Typen formalisiert.

$$\frac{\begin{array}{c}\text{(C,D)} \in \text{(sub\_class1 G)}* \qquad \text{is\_bclass G C} \mid \text{is\_tclass G C} \qquad \text{i} = \text{TheVoid} \\ \text{is\_bclass G C} \longrightarrow \text{is\_bclass G D} \\ \text{is\_tclass G C} \longrightarrow (\text{is\_tclass G D} \mid (\text{is\_bclass G D} \,\&\, \text{D} = \text{Object}))\end{array}}{(\text{Class i D}, \text{Class i C}) \in \text{(narrow G)}}$$

$$\frac{\begin{array}{c}\text{(C,D)} \in \text{(sub\_class1 G)}* \qquad \text{is\_rclass G C} \mid \text{is\_bclass G C} \qquad \text{i} = \text{TThis} \\ \text{is\_rclass G C} \longrightarrow (\text{is\_rclass G D} \mid \\ \text{is\_bclass G D} \,\&\, (\exists\, \text{B}.\ (\text{B,D}) \in (\text{sub\_class1 G})^+ \,\&\, \text{is\_rclass G B}) \\ \text{is\_bclass G C} \longrightarrow \text{is\_bclass G D} \,\&\, (\exists\, \text{B}.\ (\text{B, D}) \in (\text{sub\_class1G})^+ \,\&\, \text{is\_rclass G B})\end{array}}{(\text{Class i D}, \text{Class i C}) \in \text{(narrow G)}}$$

6.2.3 Cast-Relation zwischen statischen instanzbasierten Typen

Die Cast-Relation zwischen statischen instanzbasierten Typen zur Konvertierung von Typen ist mit Hilfe der Weitungs- und Einengungsrelation zwischen statischen instanzbasierten Typen modelliert.

$$\frac{(\text{S, T}) \ \in \ \text{(widen G)}}{(\text{S, T}) \ \in \ \text{(cast G)}}$$

$$\frac{(\text{S, T}) \ \in \ \text{(narrow G)}}{(\text{S, T}) \ \in \ \text{(cast G)}}$$

6.2.4 Weitungsrelation zwischen dynamischen instanzbasierten Typen

Die syntaktische Subtypbeziehung zwischen dynamischen instanzbasierten Typen ist analog zur Weitungsrelation zwischen statischen instanzbasierten Typen definiert. Sie unterscheidet sich lediglich hinsichtlich der Verwendung von dynamischen Instanzkontexten zur Bildung von dynamischen instanzbasierten Typen von der Subtyprelation zwischen statischen instanzbasierten Typen.

```
consts
  dyn_widen :: prog ⇒ (dyn_ty * dyn_ty)set
```

$$\frac{}{(\text{T, T}) : \text{dyn\_widen G}}$$

$$\frac{}{(\text{DynNT, DynRefTR}) \in \text{dyn\_widen G}}$$

$$\frac{\begin{array}{c}\text{(C,D)} \in \text{(sub\_class1 G)}* \qquad \text{is\_bclass G C} \mid \text{is\_tclass G C} \qquad \text{i} = \text{DynTheVoid} \\ \text{is\_bclass G C} \longrightarrow \text{is\_bclass G D} \\ \text{is\_tclass G C} \longrightarrow (\text{is\_tclass G D} \mid (\text{is\_bclass G D} \,\&\, \text{D} = \text{Object}))\end{array}}{(\text{DynClass i C}, \text{DynClass i D}) \in \text{(dyn\_widen G)}}$$

$$\frac{\begin{array}{c}\text{(C,D)} \in \text{(sub\_class1 G)}* \qquad \text{is\_rclass G C} \mid \text{is\_bclass G C} \qquad \text{i} = \text{DynTId I} \\ \text{is\_rclass G C} \longrightarrow (\text{is\_rclass G D} \mid \\ \text{is\_bclass G D} \,\&\, (\exists\, \text{B}.\ (\text{B,D}) \in (\text{sub\_class1 G})^+ \,\&\, \text{is\_rclass G B}) \\ \text{is\_bclass G C} \longrightarrow \text{is\_bclass G D} \,\&\, (\exists\, \text{B}.\ (\text{B, D}) \in (\text{sub\_class1G})^+ \,\&\, \text{is\_rclass G B})\end{array}}{(\text{DynClass i C}, \text{DynClass i D}) \in \text{(dyn\_widen G)}}$$

Entsprechend der Definition des Prädikats widens ist das Prädikat dyn_widens zur Überprüfung modelliert, ob die Typen zweier Listen dynamischer instanzbasierter Typen paarweise zueinander weitbar sind.

```
constdefs
  dyn_widens :: prog ⇒ [dyn_ty list, dyn_ty list] ⇒ bool
  dyn_widens G Ts Ts' == list_all2 (λT T'. (T,T') ∈ (dyn_widen G)) Ts Ts'
```

Die Transitivität der Weitungsrelation zwischen dynamischen instanzbasierten Typen wird analog zum Nachweis der Transitivität der Subtyprelation zwischen statischen instanzbasierten Typen durch den Nachweis des Lemmas dyn_widen_trans gezeigt.

```
lemma dyn_widen_trans [rule_format]:
  [|(S,U):(dyn_widen G); ws_prog G|]
  ⟹ ∀ T. (U,T) ∈ (dyn_widen G) ⟶ (S,T) ∈ (dyn_widen G)
```

6.3 Wohlgetyptheit von Termen

Die Wohlgetyptheit von Termen wird durch Typisierungsregeln modelliert. Die Typisierung von Termen findet relativ zu einem statischen Environment statt. Dieses ist um den Teamkontext erweitert, abhängig von dem statische Typen von Termen bestimmt werden. Die Typisierungsregeln sind implizit mit dem Teamkontext parametrisiert, der eine Eingrenzung der tatsächlichen Typen von Termen ermöglicht. Die innerhalb der Typisierungsregeln formulierten Einschränkungen stellen die während eines *Typechecking*-Prozesses durchgeführten Überprüfungen dar.

6.3.1 Typisierung relativ zum statischen Environment

Das statische Environment ist durch das Typsynonym *env* formalisiert. Es besteht zum einen aus einem Programm, in dem die zu typisierenden Terme enthalten sind. Innerhalb des Kompiliervorgangs stellt es gerade das Programm dar, dessen Wohlgeformtheit überprüft werden soll. Zum anderen besteht das statische Environment aus einem bei einem Methodenaufruf gebildeten *invocation frame*, der durch den Recordtyp inv_frame modelliert ist. Motivation für die Einführung des Typs inv_frame ist die Realisierung einer für Object Teams spezifischen Einschränkung beim Aufruf redefinierter Methoden entlang der impliziten Vererbungshierarchie (siehe Abschnitt 6.3.4). Daher wird der Name der Methode, aus der ein zu typisierender Term stammt, als Feld des Recordtyps inv_frame mit dem Namen act_method modelliert. Der Name der Klasse, in der die Methode definiert ist, die einen zu typisierenden Term enthält, ist als Feld des Recordtyps inv_frame mit dem Namen act_class modelliert. Dadurch können Überprüfungen bei der Typisierung von Termen zum Beispiel innerhalb der Konstanten check_tcontext (siehe Abschnitt 6.3.3) unabhängig von der Ausführung von statischen oder Instanzmethoden modelliert werden. Das Feld des Recordtyps inv_frame mit dem Namen lenv dient zur Modellierung des statischen lokalen Environments. Dieses ist als *table* formalisiert und ordnet Methodenparametern, lokalen Variablen und This deren statischen Typ zu. Der der modellierungsspezifischen Variablen TContext (siehe Abschnitt 5.1.1) zugeordnete Typ stellt den aktuellen Teamkontext des statischen Environments dar. Dieser wird zur Bindung von Rollentypen verwendet. Bei der Überprüfung der Wohlgetyptheit eines Methodenkörpers oder -rückgabeausdrucks bei der Wohlgeformtheitsprüfung eines ObjectTeams/Java-Programms wird der Variablen TContext der durch die Teamklasse gebildete Typ zugewiesen, in der der Teammethodenkörper oder -rückgabeausdruck definiert ist, beziehungsweise der Typ, der durch die die Rollenklasse umschließende Teamklasse gebildet ist, in der Rollenmethodenkörper oder -rückgabeausdruck definiert ist. Zur Laufzeit können Methoden auf Team- und Rollenobjekten ausgeführt werden, die Instanzen von Klassen einer spezielleren Teamschicht darstellen. Während der statischen Analyse werden Rollentypen nur eingegrenzt. Erst bei der Typisierung von ObjectTeams/Java-Konstrukten relativ zur Programmausführung werden sie abhängig vom dynamischen Typ des Teamobjekts exakt bestimmt, auf dem eine Methode ausgeführt wird, beziehungsweise abhängig vom dynamischen Typ des Teamobjekts, in dem das Rollenobjekt enthalten ist, auf dem eine Methode ausgeführt wird.

```
types env = prog * inv_frame

record inv_frame =
  act_class :: tname
  act_method :: mname
  lenv :: vname ⇒ ty option
```

Folgende syntaktische Abkürzungen werden zum Zugriff auf die Bestandteile des statischen Environments definiert. Die syntaktische Abkürzung prg modelliert den Zugriff auf das Programm des statischen En-

vironments, die Abkürzung lcl den Zugriff auf das lokale statische Environment, die Abkürzung tcontext den Zugriff auf den Teamkontext, die Abkürzung act_cls den Zugriff auf den aktuellen Klassennamen und die Abkürzung act_m den Zugriff auf den Namen der Methode, in der ein zu typisierender Term enthalten ist.

```
syntax
    prg :: env ⇒ prog
    lcl :: env ⇒ (vname ⇒ ty option)
    tcontext :: env ⇒ ty option
    act_cls :: env ⇒ tname
    act_m :: env ⇒ mname

translations
    prg E ⇒ fst E
    lcl E == lenv (snd E)
    tcontext E == (lenv (snd E)) TContext
    act_cls E == act_class (snd E)
    act_m E == act_method (snd E)
```

6.3.2 Überprüfung und Bindung von Rollentypen relativ zum Teamkontext

Rollenklassen werden bei der Bildung von Subteamklassen redefiniert, so dass im Kontext einer Teamklasse jeweils nur in dieser enthaltene Rollenklassen sichtbar sind. Dies hat für die Modellierung einer Wohlgetyptheitsüberprüfung von Termen zwei Konsequenzen.

1. In einer Teamklasse und den darin enthaltenen Rollenklassen dürfen nur Rollentypen deklariert und denotiert werden, die durch Rollenklassen der Teamklasse oder Superbasisklassen dieser Rollenklassen gebildet sind. Denn nur diese Rollentypen besitzen alle Eigenschaften, die im Kontext dieser Teamklasse erwartet werden.

   Diese Einschränkung wird durch folgende Konstanten überprüft.

   - Typen von Attributen, Methodenparametern und lokalen Variablen werden innerhalb des Wohlgeformtheitsprädikats wf_ty_decl (siehe Abschnitt 7.1.1) überprüft.
   - Typen innerhalb von Methodenkörpern in den Ausdrücken NewC, Cast und Inst werden innerhalb der Konstanten check_type (siehe Abschnitt 6.3.3) überprüft.

2. Rollentypen werden relativ zum Teamkontext des statischen Environments gebunden. Die folgenden Konstanten sind zur Bindung von Rollentypen relativ zum Teamkontext des statischen Environments und zur Überprüfung von Typen modelliert.

   - Die Bindung von in den Ausdrücken NewC, Cast und Inst denotierten Typen und die Bindung der Typen der Ausdrücke Super, TSuper und this ist durch die Konstante check_type (siehe Abschnitt 6.3.3) modelliert.
   - Die Bindung von Typen von Attributen und Methodenparametern ist durch die Konstante adjust_t (siehe Abschnitt 6.3.3) modelliert und findet bei der Typisierung der Ausdrücke FAcc und Call statt.
   - Die Bindung von Typen von Variablen des statischen lokalen Environments ist durch die Konstante adjust_t modelliert (siehe Abschnitt 7.1.3). Die Überprüfung der Typen von Variablen des statischen lokalen Environments ist durch die Konstante check_local_type (siehe Abschnitt 6.3.3) modelliert und findet bei der Typisierung des Ausdrucks LAcc statt.

6.3.3 Konstanten zum Binden von Rollentypen relativ zum Teamkontext

Konstanten zur Bestimmung von im Teamkontext sichtbaren Rollenklassen Die Konstante rrc (*redefining role class*) modelliert die Bestimmung der Rollenklasse der Teamklasse TC relativ zum Programm G, die die Rollenklasse R redefiniert. Dabei wird die Konstante crrc (*calculate redefining role class*) verwendet. Diese berechnet die die Rollenklasse R entlang der impliziten Vererbungshierarchie redefinierende Rollenklasse aus der Liste der Rollenklassen der Teamklasse TC. Die Konstanten rrc und crrc

sind als „partielle“ Funktionen durch Einpacken der Rückgabetypen *tname* in den Typ *option* modelliert. Denn erstens setzt ein definierter Zugriff auf die innerhalb einer Teamklassendeklaration enthaltene Rollenklassenliste die Existenz der Teamklasse TC im Programm G voraus. Zweitens wird bei der Typisierung des Ausdrucks Call der aktuelle Teamkontext indirekt an die Konstante crrc übergeben. Es findet keine Aktualisierung des Teamkontextes bei der Typisierung eines Methodenaufrufausdrucks während der statischen Analyse statt. Deshalb schließt die Definition der Konstante crrc den Fall mit ein, dass eine Teammethode mit Rollentypen in Methodenparametern auf einem Ausdruck aufgerufen wird, dessen Typ nicht zum aktuellen Teamkontext konform ist. Die Konstante rrc wird bei der Definition der Konstanten check_type, check_local_type und adjust_t verwendet.

```
consts
  rrc :: prog ⇒ tname ⇒ tname ⇒ tname option
  crrc :: tname list ⇒ prog ⇒ tname ⇒ tname ⇒ tname option
defs
  rrc_def:
  rrc G TC R ==
    if (is_tclass G TC & is_rclass G R)
    then crrc (roles (the (tclass G TC))) G TC R else None

primrec
  crrc_Nil:
  crrc [ ]G TC R = None
  crrc_Cons:
  crrc (vc # vs) G TC R =
    (if ((vc,R) ∈ (sub_vclass1 G)* & is_tclass G TC & vc ∈ set (roles (the (tclass G TC))))
    then Some vc else crrc vs G TC R)
```

Die folgenden Lemmata werden innerhalb des Nachweises der statischen Schichtenaussage bezüglich der Wohlkonstruiertheit von Typen einschließlich des Vorkommens von Rollentypen in Programmen (siehe Abschnitt 7.2) und innerhalb der Betrachtungen einer Typzuverlässigkeitsaussage (siehe Abschnitt 9.2) verwendet.

```
lemma crrc_lemma [rule_format]:
  crrc Rs G TC R = Some C ⟶ C ∈ set Rs

lemma crrc_in_sub_vclass [rule_format]:
  crrc Rs G TC R = Some C ⟶ (C,R) ∈ (sub_vclass1 G)*

lemma crrc_is_rclass:
  [|crrc Rs G TC R = Some C; is_rclass G R|] ⟹ is_rclass G C

lemma crrc_eq [rule_format]:
  ws_prog G ⟶ is_rclass G C ⟶ C ∈ set Rs ⟶ distinct Rs ⟶
    is_tclass G TC ⟶ (set Rs <= set (roles (the (tclass G TC)))) ⟶ crrc Rs G TC C = Some C

lemma rrc_lemma:
  rrc G TC R = Some C ⟹ C ∈ set (roles (the (tclass G TC)))

lemma rrc_rrc_is_rclass:
  rrc G TC R = Some C ⟹ is_rclass G C

lemma rrc_eq:
  [|ws_prog G; is_rclass G C; C ∈ set (roles (the (tclass G TC))); is_tclass G TC|]
  ⟹ rrc G TC C = Some C
```

Konstanten zur Überprüfung und Bindung von Typen Die Konstante check_type modelliert die Überprüfung von in ObjectTeams/Java-Programmen denotierten Typen und bindet Rollentypen relativ zum akutellen Teamkontext. Sie wird bei der Typisierung der Ausdrücke NewC, Cast, Inst, Super, TSuper und this verwendet (siehe Abschnitt 6.3.4). Die Konstante check_type ist zusammen mit der Konstanten check_rtype mittels primitver Rekursion über die Typen *ty* und *ref_ty* definiert.

```
consts
  check_rtype :: env ⇒ ref_ty ⇒ ty option
  check_type :: env ⇒ ty ⇒ ty option
primrec
  check_type_NT:
  check_rtype E NullT = None
```

Der Null-Typ wird auf None abgebildet, da er bei der Denotation und Deklaration von Typen nicht verwendet werden darf [GJS+00] (§ 4.1). Bezüglich eines Klassentyps wird überprüft, ob dieser durch eine im Programm prg E sichtbare Klasse und durch einen entsprechenden Instanzkontext gebildet ist.

- Ein durch eine Basisklasse gebildeter Typ darf um den Instanzkontext TheVoid angereichert sein oder um den Instanzkontext TThis. Wenn er um den Instanzkontext TThis angereichert ist, dann muss die Basisklasse eine Superklasse einer im aktuellen Teamkontext enthaltenen Rollenklasse darstellen, und der aktuelle Teamkontext muss den innerhalb der Konstanten check_tcontext und roles_tcontext definierten Einschränkungen genügen.

  Die Konstanten check_tcontext und roles_tcontext werden im folgenden Abschnitt erläutert.

- Ein durch eine Teamklasse gebildeter Typ muss um den Instanzkontext TheVoid angereichert sein.

- Ein durch eine Rollenklasse gebildeter Typ muss um den Instanzkontext TThis angereichert sein, der aktuelle Teamkontext muss den innerhalb der Konstanten check_tcontext und roles_tcontext definierten Einschränkungen genügen, und der Rollentyp muss innerhalb einer Instanzmethode vorkommen.

  Folgende vier Fälle werden bei der Überprüfung und Bindung eines Rollentyps unterschieden.

  1. Wenn der Rollentyp innerhalb einer Rollenmethode vorkommt, der Typ von this also durch eine Rollenklasse gebildet wird, und diese eine Rollenklasse des aktuellen Teamkontextes ist, dann muss auch die den zu überprüfenden Typ bildende Rollenklasse eine des aktuellen Teamkontextes sein.
  2. Wenn der Rollentyp innerhalb einer Rollenmethode vorkommt, und die Rollenklasse, in der die Methode definiert ist, nicht im aktuellen Teamkontext vorkommt, dann wird die den zu überprüfenden Typ bildende Rollenklasse relativ zum Teamkontext unter Anwendung der Konstante rrc gebunden.
  3. Wenn der Rollentyp innerhalb einer Teammethode vorkommt, der Typ von this also durch eine Teamklasse gebildet wird, und diese gleich dem akutellen Teamkontext ist, dann muss die den zu überprüfenden Typ bildende Rollenklasse in dieser enthalten sein.
  4. Wenn der Rollentyp innerhalb einer Teammethode vorkommt, und die den Typ von this bildende Teamklasse eine Superklasse des aktuellen Teamkontextes darstellt, dann wird die den zu überprüfenden Typ bildende Rollenklasse relativ zum aktuellen Teamkontext unter Anwendung der Konstante rrc gebunden.

```
check_type_Class:
  check_rtype E (ClassT i C) =
    (if (is_bclass (prg E) C & i = TheVoid) then Some (Class TheVoid C)
    else
      (if (is_bclass (prg E) C & i = TThis)
      then (if (check_tcontext E ≠ None & roles_tcontext E ≠ None &
                (∃ R ∈ set (the (roles_tcontext E)). (R,C)∈(sub_class1 (prg E))^+))
            then Some (Class TThis C) else None)
      else
        (if (is_tclass (prg E) C & i = TheVoid) then Some (Class TheVoid C)
        else (if (is_rclass (prg E) C & i = TThis)
              then (if (check_tcontext E ≠ None & roles_tcontext E ≠ None & cls_This E ≠ None)
                    then (if (is_rclass (prg E) (the (cls_This E)))
                          then (if (the (cls_This E) ∈ set (the (roles_tcontext E)))
                                then (if (C ∈ set (the (roles_tcontext E)))
                                      then Some (Class TThis C) else None)
```

```
                    else (if (rrc (prg E) (the (check_tcontext E)) C ≠ None)
                        then Some (Class TThis (the (rrc (prg E) (the (check_tcontext E)) C)))
                        else None))
            else
             (if (is_tclass (prg E) (the (cls_This E)))
             then (if (cls_This E = check_tcontext E)
                   then (if (C ∈ set (the (roles_tcontext E)))
                         then Some (Class TThis C) else None)
                   else (if (rrc (prg E) (the (check_tcontext E)) C ≠ None)
                       then Some (Class TThis (the (rrc (prg E) (the (check_tcontext E)) C)))
             else None)) else None)) else None) else None))))
```

```
primrec
  check_type_PrimT:
  check_type E (PrimT pt) = Some (PrimT pt)
  check_type_RefT:
  check_type E (RefT rt) = check_rtype E rt
```

Primitive Typen können beliebig im Programmtext denotiert werden.

Die Konstante check_tcontext modelliert die Überprüfung und den Zugriff auf den im statischen Environment E enthaltenen Teamkontext abhängig von der Art der aktuellen Klasse des Environments (act_cls E).

- Wenn es sich bei der aktuellen Klasse des Environments um eine Basisklasse handelt, dann darf kein Teamkontext gesetzt sein. Um eine einheitliche Behandlung beim Zugriff auf den Teamkontext zum Beispiel innerhalb der Typisierungsregeln für Attributzugriffe und Methodenaufrufe zu ermöglichen, wird anstelle von None die aktuelle Klasse des statischen Environments zurückgeben, die jedoch nicht zur Bindung von Rollenklassen verwendet wird, da innerhalb der Definition der Konstante rrc überprüft wird, ob sie eine Teamklasse darstellt.
- Wenn es sich bei der aktuellen Klasse des statischen Environments um eine Teamklasse handelt, der Zugriff auf den Teamkontext innerhalb einer Instanzmethode stattfindet, und die aktuelle Klasse des statischen Environments höchstens allgemeiner als die den Typ des im Environment enthaltenen Teamkontextes bildende Teamklasse ist, dann wird letztere zurückgegeben.
- Wenn es sich bei der aktuellen Klasse des Environments um eine Rollenklasse handelt, der Zugriff auf den Teamkontext innerhalb einer Instanzmethode stattfindet, und die die aktuelle Klasse des statischen Environments umschließende Teamklasse höchstens allgemeiner als die den Typ des im Environment enthaltenen Teamkontextes bildende Teamklasse ist, dann wird letztere zurückgegeben.

```
constdefs
  check_tcontext :: env ⇒ tname option
  check_tcontext E ==
    (if (is_bclass (prg E) (act_cls E) & tcontext E = None)
    then Some (act_cls E)
    else (if (is_tclass (prg E) (act_cls E) & tcontext E ≠ None &
              cls_This E ≠ None & cls (the (tcontext E)) ≠ None &
              is_tclass (prg E) (the (cls (the (tcontext E)))) &
              (the (cls (the (tcontext E))), act_cls E) ∈ (sub_class1 (prg E))*)
          then cls (the (tcontext E))
          else (if (is_rclass (prg E) (act_cls E) & tcontext E ≠ None &
                    cls_This E ≠ None & cls (the (tcontext E)) ≠ None &
                    is_tclass (prg E) (the (cls (the (tcontext E)))) &
                    (the (cls (the (tcontext E))), encl_class (the (rclass (prg E) (act_cls E)))) ∈
                     (sub_class1 (prg E))*)
                then cls (the (tcontext E)) else None)))
```

Die Konstante cls_This liefert die den Typ von this bildende Klasse. Innerhalb der Definition der Konstante wird überprüft, ob der Typ von this ein relativ zum Programm prg E korrekt gebauter Klassentyp

ist durch Anwendung des Prädikats is_class_type, und ob die aktuelle Klasse des Environments (act_cls E) gleich der Klasse ist, die in Typ von this bildet.

```
constdefs
  cls_This :: env ⇒ tname option
  cls_This E ==
    if ((lcl E) This = None) then None
    else (if (is_class_type (prg E) (the ((lcl E) This)) & the (cls (the ((lcl E) This))) = act_cls E)
        then cls (the ((lcl E) This)) else None)
```

Das Prädikat is_class_type überprüft, ob es sich bei dem übergebenen Typ um einen handelt, der durch eine im Programm G sichtbare Klasse und durch einen entsprechenden Instanzkontext gebildet ist - also bei einer Basisklasse durch TheVoid oder TThis, bei einer Teamklasse durch TheVoid und bei einer Rollenklasse durch TThis. Durch Anwendung des Prädikats ic_correct wird die Bildung des Typs weiter eingeschränkt, weil gefordert wird, dass der durch eine Basisklasse gebildete Typ um den Instanzkontext TheVoid angereichert ist, da der Typ von this kein Rollentyp sein darf, wenn er durch eine Basisklasse gebildet wird. Das Prädikat is_class_type ist zusammen mit dem Prädikat is_class_rtype mittels primitiver Rekursion über die Typen ty und ref_ty modelliert.

```
consts
  is_class_rtype :: prog ⇒ ref_ty ⇒ bool
  is_class_type :: prog ⇒ ty ⇒ bool
primrec
  is_class_type_NT:
  is_class_rtype G (NullT) = False
  is_class_type_Class:
  is_class_rtype G (ClassT i C) =
      (((is_bclass G C | is_tclass G C) & i = TheVoid) |
      ((is_bclass G C | is_rclass G C) & i = TThis))
primrec
  is_class_type_PrimT:
  is_class_type G (PrimT pt) = False
  is_class_type_RefT:
  is_class_type G (RefT rt) = is_class_rtype G rt
```

Das innerhalb der Konstante is_class_rtype und innerhalb der Regel zur Typisierung des Ausdrucks NewC verwendete Prädikat ic_correct dient zur Modellierung der Überprüfung, ob der Instanzkontext TheVoid in Kombination mit einer Basis- oder Teamklasse C und der Instanzkontext TThis zusammen mit einer Rollenklasse C auftritt.

```
constdefs
  ic_correct :: prog ⇒ ic ⇒ tname ⇒ bool
  ic_correct G i C ==
    (case i of
      TheVoid ⇒ (is_bclass G C | is_tclass G C)
      | TThis ⇒ is_rclass G C)
```

Die Konstante roles_tcontext modelliert die Überprüfung des Teamkontextes und den Zugriff auf die im Teamkontext des statischen Environments sichtbaren Rollenklassen. Es wird überprüft, ob der Teamkontext des statischen Environments den innerhalb der Konstanten check_tcontext modellierten Einschränkungen genügt, und ob er eine Teamklasse darstellt.

```
constdefs
  roles_tcontext :: env ⇒ (tname list) option
  roles_tcontext E ==
    if (check_tcontext E ≠ None & is_tclass (prg E) (the (check_tcontext E)))
    then Some (roles (the (tclass (prg E) (the (check_tcontext E))))) else None
```

Die Konstante check_local_type dient im Gegensatz zu den Definitionen der Konstanten check_type und check_rtype zur Überprüfung, ob Rollentypen ausschließlich durch im aktuellen Teamkontext enthaltene

Rollenklassen beziehungsweise Superbasisklassen von im aktuellen Teamkontext enthaltenen Rollenklassen gebildet werden. Diese striktere Forderung an im statischen lokalen Environment enthaltene Typen entspricht der in [KNO+02a] und [KNO+02b] formulierten Überprüfung, ob die im statischen lokalen Environment enthaltenen Typen durch im Programm (prg E) sichtbare Klassen gebildet sind. Die Forderung dieser Einschränkung bei der Typisierung von Variablen des statischen lokalen Environments ist notwendig, weil über das statische lokale Environment während der Typisierung von Termen keine Informationen bekannt sind. Die Definition eines neuen Typs zur Formalisierung des statischen lokalen Environments inklusive der Forderung, dass alle im statischen lokalen Environment enthaltenen Typen korrekt gebildet sind, würde den Bau von statischen lokalen Environments nicht wohlgeformter Programme unmöglich machen. Deshalb werden die Typen des statischen lokalen Environments innerhalb der Typisierungsregeln überprüft und nicht innerhalb der Definition eines Typs zur Modellierung von statischen lokalen Environments. Die Konstante check_local_type ist zusammen mit der Konstanten check_local_rtype mittels primitiver Rekursion über die Typen ty und ref_ty definiert.

```
consts
  check_local_rtype :: env ⇒ ref_ty ⇒ ty option
  check_local_type :: env ⇒ ty ⇒ ty option
primrec
  check_local_type_NT:
  check_local_rtype E NullT = None
  check_local_type_Class:
  check_local_rtype E (ClassT i C) =
    (if (is_bclass (prg E) C & i = TheVoid) then Some (Class TheVoid C)
    else (if (is_bclass (prg E) C & i = TThis)
        then (if (check_tcontext E ≠ None & roles_tcontext E ≠ None &
                (∃ R ∈ set (the (roles_tcontext E)). (R,C) ∈ (sub_class1 (prg E))^+))
             then Some (Class TThis C) else None)
        else (if (is_tclass (prg E) C & i = TheVoid)
            then Some (Class TheVoid C)
            else (if (is_rclass (prg E) C & i = TThis)
                then (if (check_tcontext E ≠ None & roles_tcontext E ≠ None &
                          cls_This E ≠ None & C ∈ set (the (roles_tcontext E)))
                     then Some (Class TThis C) else None)
                else None))))
primrec
  check_local_type_PrimT:
  check_local_type E (PrimT pt) = Some (PrimT pt)
  check_local_type_RefT:
  check_local_type E (RefT rt) = check_local_rtype E rt
```

Das Binden von Rollentypen relativ zum aktuellen Teamkontext beim Zugriff auf ein Attribut und beim Aufruf einer Methode wird durch die mittels primitiver Rekursion über die Typen ref_ty und ty definierten Funktionen adjust_rt und adjust_t realisiert.

Die Konstante adjust_rt überprüft auch den Instanzkontext von Typen und bezüglich durch Basisklassen gebildeten Rollentypen, ob die Basisklassen Superklassen von im Teamkontext des statischen Environments enthaltenen Rollenklassen darstellen. Diese Überprüfungen waren beim Entwurf der Konstanten nicht vorgesehen. Zum Abschluss der statischen Analyse von ObjectTeams/Java wurde versucht, eine Aussage über das modellierte Typsystem von ObjectTeams/Java nachzuweisen. Diese in Abschnitt 7.2 formulierte statische Schichtenaussage bringt zum Ausdruck, dass bei der Typisierung eines beliebigen Terms relativ zum statischen Environment ein Typ bestimmt wird, der wohlkonstruiert ist (siehe Abschnitt 7.2). Wenn es sich bei diesem Typ um einen Rollentyp handelt, dann soll gelten, dass dieser durch eine im Teamkontext des statischen Environments sichtbare Rollenklasse gebildet ist, oder durch eine Superbasisklasse einer solchen Rollenklasse. Zum Nachweis dieser Aussage werden Typen also sowohl hinsichtlich ihrer Bestandteile überprüft, als auch Rollentypen hinsichtlich ihrer Bindung. Für den Nachweis, dass Typen von Attributzugriffs- und Methodenaufrufsausdrücken wohlkonstruiert sind, werden diese Einschränkungen innerhalb eines in Abschnitt 7.1.1 definierten Wohlgeformtheitsprädikats bezüglich Typdeklarationen überprüft. Die Wohlgeformtheit von Attribut- und Methodendeklarationen

wird nur für den Kontext überprüft, in dem die Attribute und Methoden definiert sind. Für den Nachweis der Schichtenaussage wird jedoch eine Aussage über die Wohlgeformtheit von Attribut- und Methodendeklarationen in allen Kontexten benötigt, in denen diese wiederverwendet werden können. Beim Nachweis der statischen Schichtenaussage wurde noch nicht erkannt, dass dafür eine Aussage über die modulare Typüberprüfung von ObjectTeams/Java-Programmen nötig ist. Diese Problematik wird in Abschnitt 7.1.5 diskutiert. Deshalb wurde die Konstante adjust_rt zum Nachweis der statischen Schichtenaussage um die Überprüfungen hinsichtlich des Instanzkontextes eines Typs und die Überprüfung, ob eine Basisklasse eines Rollentyps Superbasisklasse einer Rollenklasse des akutellen Teamkontextes ist, erweitert.

```
consts
  adjust_rt :: prog ⇒ tname ⇒ ref_ty ⇒ ty option
  adjust_t :: prog ⇒ tname ⇒ ty ⇒ ty option

primrec
  adjust_t_NT:
  adjust_rt G TC NullT = Some (NT)
  adjust_t_Class:
  adjust_rt G TC (ClassT i C) =
    (if ((is_bclass G C | is_tclass G C) & i= TheVoid) then Some (Class TheVoid C)
    else (if (is_bclass G C & i=TThis & is_tclass G TC &
            (∃ R ∈ set (roles (the (tclass G TC))). (R,C) ∈ (sub_class1 G)^+))
          then Some (Class TThis C)
          else (if (is_rclass G C & i=TThis)
                then (if (is_tclass G TC & rrc G TC C ≠ None)
                      then Some (Class i (the (rrc G TC C))) else None)
                else None)))

primrec
  adjust_t_PrimT:
  adjust_t G TC (PrimT pt) = Some (PrimT pt)
  adjust_t_RefT:
  adjust_t G TC (RefT rt) = adjust_rt G TC rt
```

Angewendet wird die Funktion adjust_t innerhalb der Definitionen der Konstanten adjust_f, adjust_s und adjust_m zum Binden von in Attribut- und Methodendeklarationen vorkommenden Rollentypen relativ zum Teamkontext des statischen Environments. Typen von lokalen Variablen werden innerhalb der Konstante adjust_m nicht relativ zum Teamkontext gebunden. Denn dann müsste bereits während der statischen Analyse eine Aktualisierung des Teamkontextes beim Aufruf einer Methode modelliert werden. Da dies erst bei der Typisierung von Methoden relativ zur Ausführung erforderlich ist, werden die Typen von lokalen Variablen innerhalb der Konstanten create_inv_frame (siehe Abschnitt 7.1.3) zur Modellierung des Baus eines statischen lokalen Environments relativ zum Teamkontext, zu dem die Methode ausgeführt wird, gebunden.

```
consts
  adjust_f :: prog ⇒ tname ⇒ field ⇒ field option
  adjust_s :: prog ⇒ tname ⇒ ty list ⇒ ty list option
  adjust_m :: prog ⇒ tname ⇒ methd ⇒ methd option

  defs
  adjust_f_def:
  adjust_f G TC f ==
    if (adjust_t G TC (field_ty f) ≠ None)
    then Some (f (|field_ty := the (adjust_t G TC (field_ty f))|)) else None

  adjust_s_def:
  adjust_s G TC pTs ==
    if (∀ p ∈ set pTs. adjust_t G TC p ≠ None)
    then Some (map (λp. (the (adjust_t G TC p))) pTs) else None
```

```
adjust_m_def:
  if (adjust_t G C TC (res_ty m) ≠ None)
  then Some (m (|res_ty := the (adjust_t G TC (res_ty m))|)) else None
```

Beim Zugriff auf Attribute wird die Konstante adjust_f innerhalb der Definitionen der Konstanten cfields_ex und cfields_v angewendet. Diese modellieren den Zugriff auf Attribute entlang der `extends`-Vererbungshierarchie - gekennzeichnet durch die Endung _ex des Konstantennamens - und der impliziten Vererbungshierarchie - gekennzeichnet durch die Endung _v des Konstantennamens. Die Konstanten cfields_ex und cfields_v liefern eine Liste aller entlang der `extends`- und der impliziten Vererbungshierarchie geerbten und um den Namen der das Attribut definierenden Klasse angereicherten Attribute einer Klasse C. Sie werden bei der Erzeugung von Objekten, die Felder für alle geerbten und in der instanziierten Klasse definierten Attribute besitzen, verwendet. Demgegenüber modellieren die Konstanten cfield_ex und cfield_v den Zugriff auf ein Attribut über den Attributnamen abhängig vom statischen Typ des Ausdrucks, über den auf das Attribut zugegriffen wird. Gleichnamige Attribute aus Superklassen der Klasse, die den statischen Typ des Ausdrucks bildet, können überdeckt werden. Deshalb ist der Rückgabetyp der Konstanten cfield_ex und cfield_v als *table* modelliert.

```
consts
  cfields_ex :: prog ⇒ tname ⇒ tname ⇒ ((vname * tname) * field option) list
  cfields_v :: prog ⇒ tname ⇒ tname ⇒ ((vname * tname) * field option) list
  cfield_ex :: prog ⇒ tname ⇒ tname ⇒ (vname, tname * field option) table
  cfield_v :: prog ⇒ tname ⇒ tname ⇒ (vname, tname * field option) table

defs
  cfields_ex_def:
  cfields_ex G C TC ==
    (map (λ((fn,fd),f). ((fn,fd),adjust_f G TC f))
      (class_rec (G,C) [] (λC c ts. (map (λ(fn,f). ((fn,C),f)) (attrs c))  ts)))

  cfields_v_def:
  cfields_v G C TC ==
    (map (λ((fn,fd),f). ((fn,fd),adjust_f G TC f))
      (vclass_rec (G,C) [] (λC c ts. (map (λ(n,f). ((n,C),f)) (attrs c))  ts)))

  cfield_ex_def:
  cfield_ex G C TC == table_of (map (λ((fn,fd),f). (fn,(fd,f))) (cfields_ex G C TC))

  cfield_v_def:
  cfield_v G C TC == table_of (map (λ((fn,fd),f). (fn,(fd,f))) (cfields_v G C TC))
```

Die Konstante cfield vereinheitlicht den Zugriff auf Attribute entlang der `extends`- und der impliziten Vererbungshierarchie. Entlang der impliziten Vererbungshierarchie dürfen Attribute nicht überdeckt werden (siehe Abschnitt 7.1.4). Konflikte zwischen gleichnamigen entlang der `extends`- und der impliziten Vererbungshierarchie geerbten Attributen treten nicht auf, weil Attribute abhängig vom statischen Typ des Ausdrucks gebunden werden, über den auf sie zugegriffen wird.

```
consts
  cfield :: prog ⇒ tname ⇒ tname ⇒ (vname, tname * field option) table

defs
  cfield_def:
  cfield G C TC ==
    if (is_rclass G C & is_tclass G TC)
    then (cfield_ex G C TC) ++ (cfield_v G C TC)
    else (if (is_tclass G C & is_tclass G TC)
          then cfield_ex G C TC
          else (if (is_bclass G C) then cfield_ex G C TC else empty))
```

Der Zugriff auf Methoden ist analog zum Zugriff auf Attribute modelliert. Vereinfachend wird kein Überladen von Methoden betrachtet. Infolgedessen müssen alle in einer Klasse definierten Methoden einen eindeutigen Namen haben (siehe Abschnitt 7.1.3), und der Schlüssel des *table* der Konstanten cmethod_ex und cmethod_v hat im Gegensatz zu [KNO+02a] nicht den Typ *sig*, sondern den Typ *mname*.

Für die statische Analyse notwendig ist eine Bindung aller Methodenparameter einer Methode, was durch Anwendung der Konstanten adjust_s zur Bindung der Methodeneingabeparameter und adjust_m zur Bindung des Methodenrückgabetyps modelliert ist.

```
consts
  cmethods_ex :: prog ⇒ tname ⇒ tname ⇒
        (mname * tname * (ty list)option * methd option) list
  cmethods_v :: prog ⇒ tname ⇒ tname ⇒
        (mname * tname * (ty list)option * methd option) list
  cmethod_ex :: prog ⇒ tname ⇒ tname ⇒
        (mname, (tname * (ty list)option * methd option))table
  cmethod_v :: prog ⇒ tname ⇒ tname ⇒
        (mname, (tname * (ty list)option * methd option))table
  cmethod :: prog ⇒ tname ⇒ tname ⇒
        (mname, (tname * (ty list)option * methd option))table

defs
  cmethods_ex_def:
  cmethods_ex G C TC ==
    (map (λ(mn,md,pTs,m). (mn,md, adjust_s G TC pTs, adjust_m G TC m))
      (class_rec (G,C) [] (λC c ts. (map (λ(s,m).
           (sig.name s,C,sig.param_tys s,m)) (methods c))  ts)))
  cmethods_v_def:
  cmethods_v G C TC ==
    (map (λ(mn,md,pTs,m). (mn,md, adjust_s G TC pTs, adjust_m G TC m))
      (vclass_rec (G,C) [] (λC c ts. (map (λ(s,m).
           (sig.name s,C,sig.param_tys s,m)) (methods c))  ts)))
  cmethod_ex_def:
    cmethod_ex G C TC == table_of (cmethods_ex G C TC)
  cmethod_v_def:
    cmethod_v G C TC == table_of (cmethods_v G C TC)
  cmethod_def:
  cmethod G C TC ==
    if (is_rclass G C & is_tclass G TC)
    then (cmethod_ex G C TC) ++ (cmethod_v G C TC)
    else (if (is_tclass G C & is_tclass G TC)
          then cmethod_ex G C TC
          else (if (is_bclass G C) then cmethod_ex G C TC else empty))
```

6.3.4 Modellierung von Typisierungsregeln

Die Regeln zur Typisierung von Termen sind als *introduction rules* der induktiv definierten Menge *wt* formalisiert und ermöglichen eine einheitliche Typisierung von Termen. Sie sind implizit um den Teamkontext des statischen Environments parametrisiert, relativ zu dem Typen von Termen gebunden werden. Syntaktische Annotationen werden zur besseren Lesbarkeit der Typisierungsregeln definiert. Der Typ *tys* fasst den Typ *ty* und den Typ *ty list* zusammen.

```
types tys = ty + ty list

consts
  wt :: (env * term * tys) set

syntax
  wt_ :: env ⇒ [term, tys] ⇒ bool (_|=_::_ [51,51,51] 50)
  wt_stmt :: env ⇒ stmt ⇒ bool (_|=_:<> [51,51] 50)
  wt_expr :: env ⇒ [expr, ty] ⇒ bool (_|=_:-_ [51,51,51] 50)
  wt_target :: env ⇒ [target, ty] ⇒ bool (_|=_:=_ [51,51,51] 50)
  wt_exprs :: env ⇒ [expr list, ty list] ⇒ bool (_|=_:#_ [51,51,51] 50)
```

```
translations
  E |= t :: T == (E, t, T) : wt
  E |= s :<> == (E, Inl (In1 s), Inl (PrimT Void)) : wt
  E |= e :- T == (E, Inl (In2 e), Inl T) : wt
  E |= c := T == (E, Inl (In3 c), Inl T) ∈ wt
  E |= l :# T == (E, Inr l, Inr T) ∈ wt
```

Die Typisierung der Terme Skip, Expr, Comp, Cond und Loop vom Typ *stmt* und die Typisierung von Ausdruckslisten vom Typ *expr list* ist aus [KNO+02a] übernommen. Anweisungen wird der Typ PrimT Void zugewiesen. Einer Liste von Ausdrücken wird die Liste der Typen ihrer Ausdrücke zugewiesen.

$$\frac{}{E \models Skip :<>}$$

$$\frac{E \models e :- T}{E \models Expr\ e :<>}$$

$$\frac{E \models s1 :<> \qquad E \models s2 :<>}{E \models s1;;\ s2 :<>}$$

$$\frac{E \models e :- PrimT\ Boolean \qquad E \models s1 :<> \qquad E \models s2 :<>}{E \models If(e)\ s1\ Else\ s2 :<>}$$

$$\frac{E \models e :- PrimT\ Boolean \qquad E \models s :<>}{E \models While\ (e)s :<>}$$

$$\frac{}{E \models [] :\#[]}$$

$$\frac{E \models e :- T \qquad E \models es :\#\ Ts}{E \models e\#es :\#\ T\#Ts}$$

Die Regeln zur Typisierung von Termen vom Typ *expr* und *target* werden im folgenden einzeln genauer erläutert.

Der Ausdruck NewC i C wird auf den durch die Konstante check_type (siehe Abschnitt 6.3.3) bestimmten Typ typisiert. Das Prädikat class_is_abstract dient zur Modellierung von Einschränkungen bei der Instanziierung abstrakter Klassen. Abstrakte Rollenklassen können in ObjectTeams/Java im Gegensatz zu Java-Klassen instanziiert werden, wenn ihre umschließende Teamklasse als abstrakt markiert ist. Abstrakte Basis- und Teamklassen dürfen so wie abstrakte Klassen in Java nicht instanziiert werden [GJS+00] (§8.1.1.1). Überprüfungen bezüglich der Kombination der Klasse C und des Instanzkontextes i, relativ zu dem eine Instanz der Klasse C gebildet werden soll, sind durch das Prädikat ic_correct (siehe Abschnitt 6.3.3) modelliert.

$$\frac{\begin{array}{c}check\_type\ E\ (Class\ i\ C) = Some\ (Class\ i\ C') \\ \neg(class\_is\_abstract\ (prg\ E)\ C') \qquad ic\_correct\ (prg\ E)\ i\ C'\end{array}}{E \models NewC\ i\ C :-\ Class\ i\ C'}$$

```
constdefs
  class_is_abstract :: prog ⇒ tname ⇒ bool
  class_is_abstract G C ==
    (is_bclass G C ⟶ ¬ (abstract (the (bclass G C)))) &
    (is_tclass G C ⟶ ¬ (abstract (the (tclass G C)))) &
    (is_rclass G C ⟶ abstract (the (rclass G C)) ⟶
      abstract (the (tclass G (encl_class (the (rclass G C))))))
```

Der Typ des Ausdrucks Cast e T wird durch die Konstante check_type (siehe Abschnitt 6.3.3) bestimmt

und muss in einer Weitungs- oder Einengungsbeziehung zum statischen Typ S des Ausdrucks e stehen. Bei der Auswertung des Cast-Ausdrucks kann es zu einem erwarteten Typfehler kommen. Ein Typfehler tritt auf, wenn der dynamische Typ des Wertes des Ausdrucks e allgemeiner ist als der Typ T′, zu dem dieser konvertiert werden soll.

$$\frac{E \models e :- S \qquad \text{check\_type } E\ T = \text{Some } T' \qquad (S, T') \in \text{cast prg } E}{E \models \text{Cast } T\ e :- T'}$$

Der Typ des Ausdrucks Inst e T wird analog zum Typ des Cast-Ausdrucks bestimmt. Der statische Typ RefT S des Ausdrucks e ist auf Referenztypen eingeschränkt. Bei seiner Auswertung tritt kein erwarteter Typfehler auf, wenn der dynamische Typ des Wertes des Ausdrucks e allgemeiner ist als der Typ T′. Stattdessen wird das Ergebnis der Prüfung, ob der Wert des Ausdrucks e ein Element des Typs T′ ist, auf den entsprechenden booleschen Wert abgebildet. Auf den Typ dieses booleschen Wertes wird der Ausdruck Inst e T typisiert ist.

$$\frac{E \models e :- \text{RefT } S \qquad \text{check\_type } E\ \text{RefT } T = \text{Some } T' \qquad (\text{RefT } S, T') \in \text{cast } (\text{prg } E)}{E \models \text{Inst } e\ T :- \text{PrimT Boolean}}$$

Der Ausdruck Lit v modelliert den Zugriff auf ein Literal und ist auf den primitiven Typ des Wertes v typisiert. Die Konstante typeof (siehe Abschnitt 9.1) wird zur Bestimmung dieses Typs verwendet.

$$\frac{E \models \text{typeof } (\lambda v.\ \text{None})\ v = \text{Some } (\text{DynPrimT pt})}{E \models \text{Lit } v :- (\text{PrimT pt})}$$

Der Zugriff auf eine Variable v des lokalen Environments lcl E wird auf den durch die Konstante check_local_type (siehe Abschnitt 6.3.3) bestimmten Typ typisiert, wenn es sich bei der Variablen nicht um This handelt. Wenn auf This zugegriffen wird, dann darf der zu typisierende Ausdruck LAcc This nicht in einer statischen Methode enthalten sein [GJS+00] (§8.4.3.2). Diese Einschränkung ist durch das Prädikat static_method modelliert. Der Typ des Ausdrucks LAcc This wird mit Hilfe der Konstanten check_type bestimmt. Er ist zusätzlich durch das Prädikat ic_correct eingeschränkt. Die Bestimmung des Typs des Ausdrucks LAcc This mit Hilfe der Konstanten check_type ist erforderlich, da der der Variablen This zugeordnete Typ im Gegensatz zu den Typen aller anderen im statischen lokalen Environment enthaltenen Variablen nicht relativ zum Teamkontext des statischen Environments gebunden wird. Der nicht abhängig vom Teamkontext gebundene Typ von LAcc This wird für die Realisierung von `super`-und `tsuper`-Rollenmethodenaufrufen und die korrekte Bestimmung des Typs des Ausdrucks Tthis benötigt. Auf die im lokalen Environment enthaltene Variable TContext darf nicht zugegriffen werden, weil sie keine Variable darstellt, die in ObjectTeams/Java-Programmen vorkommt, sondern eine modellierungsspezifische Varialbe.

$$\frac{\begin{array}{c}(\text{lcl } E)\ v = \text{Some } T \qquad v \neq \text{TContext} \\ v = \text{This} \longrightarrow \text{check\_type } E\ T = \text{Some } T' \ \&\ T' = (\text{Class i C}) \ \& \\ \neg\ \text{static\_method } E \ \&\ \text{ic\_correct } (\text{prg } E)\ i\ C \\ v \neq \text{This} \longrightarrow \text{check\_local\_type } E\ T = \text{Some } T'\end{array}}{E \models \text{LAcc } v :- T'}$$

```
constdefs
  static_method :: env ⇒ bool
  static_method E ==
  (∃ md pTs m. cmethod (prg E) (act_cls E) (act_cls E) (act_m E) =
                Some (md, Some pTs, Some m) & static m)
```

Der Typ des Ausdrucks Tthis ist durch die die Rollenklasse umschließende Teamklasse gebildet, in der der Ausdruck Tthis vorkommt. Er wird durch die Konstante type_Tthis abhängig von dem Typ bestimmt, der der Variablen This im lokalen Environment zugeordnet ist. Der Ausdruck Tthis darf nur innerhalb von Instanzmethoden von Rollenklassen vorkommen.

$$\frac{\text{type\_Tthis } E = \text{Some } T}{E \models \text{Tthis} :- T}$$

```
constdefs
  type_Tthis :: env ⇒ ty option
```

```
type_Tthis E ==
  (if (cls_This E = None) then None
  else (if (is_rclass (prg E) (the (cls_This E)) & check_tcontext E ≠ None)
       then Some (Class TheVoid (encl_class (the (rclass (prg E) (the (cls_This E))))))
       else None))
```

Die Zuweisung eines Ausdrucks e an eine lokale Variable v ist typisierbar, wenn der Typ S des Ausdrucks e höchstens spezieller ist als der Typ T der lokalen Variablen v, der Typ S also zum Typ T geweitet werden kann, und es sich bei der Variablen v nicht um This handelt, weil der Wert von This unveränderlich ist. Der Zuweisungsausdruck v::=e wird auf den gegebenenfalls spezielleren Typ des zugewiesenen Ausdrucks S typisiert.

$$\frac{E \models LAcc\ v :- T \qquad E \models e :- S \qquad (S, T) \in (widen\ (prg\ E)) \qquad v \neq This}{E \models v ::= e :- S}$$

Der Zugriff auf das Attribut fn eines Ausdrucks e setzt voraus, dass dessen Typ durch eine Klasse gebildet ist. Der Typ des Attributzugriffausdrucks {fd}e..fn wird mit Hilfe der Konstanten cfield bestimmt, die den Namen der Klasse fd und die Attributdeklaration f des Attributs fn ausgehend von der den statischen Typ des Ausdrucks e bildenden Klasse C unter Verwendung des durch die Konstante check_tcontext (siehe Abschnitt 6.3.3) bestimmten Teamkontextes des statischen Environments ermittelt. Der Name der Klasse fd, in der das Attribut fn definiert ist, wird zur Realisierung des statischen Bindens von Attributen annotiert. Eine weitere Einschränkung bei der Typisierung eines Attributzugriffausdrucks ist durch das Prädikat field_is_accessible modelliert. Wenn der Typ des Attributs fn durch den Instanzkontext TThis angereichert ist - es sich also um einen Rollentyp handelt - und der Typ des Ausdrucks e, über den auf das Attribut zugegriffen werden soll, durch eine Teamklasse gebildet ist, dann muss es sich bei dem Ausdruck e entweder um this oder um Tthis handeln. Dadurch wird sichergestellt, dass Attribute, die Verweise auf Rollenobjekte speichern, nur innerhalb des Teamkontextes referenziert werden können, in dem die referenzierten Rollenobjekte enthalten sind.

$$\frac{\begin{array}{c}E \models e :- Class\ i\ C \qquad check\_tcontext\ E \neq None \\ field\_is\_accessible\ E\ e\ C\ (field\_ty\ f) \\ cfield\ (prg\ E)\ C\ (the\ (check\_tcontext\ E))\ fn = Some\ (fd, Some\ f)\end{array}}{E \models \{fd\}e..fn :- (field\_ty\ f)}$$

```
constdefs
  field_is_accessible :: env ⇒ expr ⇒ tname ⇒ ty ⇒ bool
  field_is_accessible E e C T ==
    (is_tclass (prg E) C & ic T ≠ None & the (ic T) = TThis) ⟶
      (e = (LAcc This) | e = Tthis)
```

Die Zuweisung eines Ausdrucks v an ein in der Klasse fd definiertes Attribut fn eines Ausdrucks e setzt voraus, dass der Typ des Ausdrucks S zum Typ des Attributs T weitbar ist. Der Zuweisungsausdruck {fd}e..fn:=v wird auf den gegebenenfalls spezielleren Typ des zugewiesenen Ausdruck S typisiert.

$$\frac{E \models \{fd\}e..fn :- T \qquad E \models v :- S \qquad (S, T) \in widen\ (prg\ E)}{E \models \{fd\}e..fn := v :- S}$$

Ein Methodenaufrufausdruck wird auf den innerhalb der Methodendefinition m deklarierten Methodenrückgabetyp res_ty m typisiert, der durch die Konstante cmethod bestimmt wird.

$$\frac{\begin{array}{c}E \models e :- Class\ i\ C \qquad E \models ps :\# pTs \\ actual\_target\ E\ e\ C = Some\ C' \\ cmethod\ (prg\ E)\ C'\ (the\ (check\_tcontext\ E))\ mn = \\ Some\ (md, Some\ pTs', Some\ m) \\ widens\ (prg\ E)\ pTs\ pTs' \qquad check\_tcontext\ E \neq None \\ is\_callable\ E\ e\ i\ C'\ mn\ md\ pTs\ m \qquad inv\_mode\ e\ m = im\end{array}}{E \models \{md, im\}e..mn(ps) :- (res\_ty\ m)}$$

Folgende Einschränkungen und Hilfsfunktionen werden bei der Typisierung eines Methodenaufrufausdrucks überprüft beziehungsweise angewendet. Die dazu verwendeten Konstanten werden im Anschluss definiert.

- Der Typ des Ausdrucks e, auf dem die Methode mit dem Namen mn aufgerufen wird, muss durch eine Klasse gebildet sein.
- Alle bei einem Methodenaufruf übergebenen Parameterausdrücke müssen wohlgetypt sein.
- Die Konstante actual_target dient zur Modellierung der Bestimmung der Klasse, von der aus die Methode, die ausgeführt werden soll, gesucht wird. Dies ist für die Realisierung von `super`- und `tsuper`-Methodenaufrufen notwendig.
- Die Typen aller Methodenparameterausdrücke müssen zu den innerhalb der Methodendeklaration definierten und gegebenenfalls innerhalb der Konstanten cmethod relativ zum Teamkontext gebundenen Typen weitbar sein.
- Der zur Bindung aller in der Methodendefinition enthaltenen Rollentypen benötigte Teamkontext wird mit Hilfe der Konstanten check_tcontext bestimmt.
- Innerhalb der Definition des Prädikats is_callable sind folgende Einschränkungen modelliert.

  Eine aufgerufene `super`-Methode darf nicht abstrakt sein [GJS+00] (§8.4.3.1). Analog dazu darf auch eine aufgerufene `tsuper`-Methode nicht abstrakt sein. Abstrakte Instanzmethoden hingegen können aufgerufen werden, weil diese nur relativ zu einer Instanz einer Klasse aufgerufen werden können. Dann jedoch muss es auch eine Implementierung der aufgerufenen Methode geben, weil eine Klasse als abstrakt markiert sein muss, sobald sie eine abstrakte Methodendeklaration enthält. Zusätzlich können in ObjectTeams/Java Instanzen von abstrakten Rollenklassen gebildet werden, weil dann deren umschließende Teamklasse als abstrakt markiert werden muss (siehe Typisierung des Ausdrucks NewC).

  Im Gegensatz zu `super`-Methodenaufrufen in Java sollen in ObjectTeams/Java entlang der impliziten Vererbungshierarchie redefinierte Methoden nur geradlienig unter Verwendung des Ausdrucks TSuper aufgerufen werden können. Das heisst, der zur Realisierung dieser Einschränkung im Aufrufrahmen einer Methode enthaltene Methodenname muss gleich dem Namen der aufgerufenen `tsuper`-Methode sein.

  Wenn ein Methodenparametertyp der aufgerufenen Methode um den Instanzkontext TThis angereichert ist - es sich also um einen Rollentyp handelt - und der Typ des Ausdrucks, auf dem die Methode aufgerufen wird, durch eine Teamklasse gebildet ist, dann muss es sich bei dem Ausdruck e entweder um this, Tthis oder Super handeln. Dadurch wird sichergestellt, dass Rollenobjekte einer Teaminstanz nicht in den Kontext einer anderen Teaminstanz gelangen und für Rollenobjekte anderer Teaminstanzen substituiert werden können. Durch diese Einschränkung wird erreicht, dass die kovariante Redefinition von Rollenklassen bei der Betrachtung von um den Instanzkontext TThis angereicherten Rollentypen nicht zu Typfehlern führt. Wenn *externalized roles* (siehe Abschnitt 2.3) modelliert werden, müsste anhand der Instanzkontexte, über die die Typen der Rollenobjekte verankert sind, überprüft werden, ob die durch die Instanzkontexte bezeichneten Variablen Referenzen auf das Teamobjekt enthalten, auf dem die Teammethode aufgerufen wird. Diese Überprüfungen sind im Rahmen dieser Arbeit nicht enthalten.

  Wenn es sich beim Typ des *target* des Methodenaufrufs um einen Rollentyp handelt, dann muss die aufgerufene Methode in einer Rollenklasse definiert sein. Das bedeutet, dass in der jetzigen Modellierung keine aus Basisklassen geerbten Methoden auf Rollenobjekten aufgerufen und wiederverwendet werden können. Dies ist zur Realisierung von *confined role objects* nötig. Denn beim Aufruf von aus Superbasisklassen geerbten Methoden findet eine implizite Weitung des Typs von this zum durch die Klasse, in der die Methode definiert ist, gebildeteten Typ statt. Dieser ist bei Basisklassen um den Instanzkontext TheVoid angereichert ist. Dies hätte zur Folge, dass Rollenobjekte durch auf Nicht-Rollentypen typisierte Variablen beliebig referenziert werden könnten.

  Eine Möglichkeit zur Wiederverwendung von aus Basisklassen geerbten Methoden, die sicherstellt, dass Referenzen auf Rollenobjekte nicht unkontrolliert aus dem Kontext des Teamobjekts heraus gereicht werden, in dem sie enthalten sind, stellt das Konzept der anonymen Methoden [BV99] dar. Es wird in Abschnitt 10.4 skizziert. Eine andere Alternative zur Wiederverwendung von aus Superbasisklassen geerbten Methoden stellt die Modellierung von opaken Rollenobjekten dar. Dabei wird eine Weitung von Rollentypen hin zu Basistypen explizit erlaubt wird, um Rollenobjekte außerhalb des Teamkontextes, in dem sie enthalten sind, durch auf Nicht-Rollentypen typisierte

Variablen referenzieren zu können. Um sicherzustellen, dass rollenspezifische Eigenschaften nicht außerhalb ihres Teamkontext verwendet und verändert werden können, soll die Redefinition von aus Superbasisklassen geerbten Methoden in Rollenklassen ausgeschlossen werden [Her03b].

- Innerhalb der Definition der Konstanten inv_mode ist die Bestimmung des Aufrufmodus einer Methode modelliert. Die Konstruktoren des Typs *inv_mode* formalisieren die in dieser Arbeit formalisierten Aufrufmodi (siehe Abschnitt 5.3). Sie dienen zur Realisierung unterschiedlicher Auswertungsstrategien bei der Ausführung von Methoden.

```
constdefs
  actual_target :: env ⇒ target ⇒ tname ⇒ tname option
  actual_target E e C ==
    if (e = TSuper)
    then (if (cls_This E ≠ None & tsuper_is_accessible (prg E) (the (cls_This E)))
          then Some (the (implicit (the (rclass (prg E) (the (cls_This E)))))) else None)
    else (if (e = Super)
          then (if (cls_This E ≠ None & super_is_accessible (prg E) (the (cls_This E)))
                then Some (super (the (class (prg E) (the (cls_This E))))) else None)
          else Some C)
```

```
constdefs
  is_callable :: env ⇒ target ⇒ ic ⇒ tname ⇒ mname ⇒ tname ⇒ (ty list) ⇒ methd ⇒ bool
  is_callable E e i C mn md pTs m ==
    ((e = TSuper | e = Super) ⟶ ¬(abstract m)) &
    (e = TSuper ⟶ mn = act_m E) &
    (is_tclass (prg E) C ⟶
      ((∃ pT ∈ set pTs. ic pT ≠ None & the (ic pT) = TThis) |
        (ic (res_ty m) ≠ None & the (ic (res_ty m)) = TThis)) ⟶
        (e = TargetExpr (LAcc This) | e = TargetExpr (Tthis) | e = Super)) &
    (is_role_type (prg E) (Class i C) ⟶ is_rclass (prg E) md)
```

```
constdefs
  inv_mode :: target ⇒ methd ⇒ inv_mode
  inv_mode e m ==
    if (static m) then Static
    else (if (e=Super) then SuperM else (if (e=TSuper) then TSuperM else IntVir))
```

Der Ausdruck Super wird auf den mittels der Konstanten check_type bestimmten und durch das Prädikat ic_correct bezüglich des Instanzkontextes eingeschränkten Typ typisiert. Die Methode, in der der Ausdruck vorkommt, darf keine statische Methode sein, und die Superklasse der den Typ von this bildenden Klasse muss im Programm prg E sichtbar und zugreifbar sein. Diese Einschränkungen werden durch die Prädikate static_method und super_is_accessible modelliert.

$$\frac{\begin{array}{ll}(\mathsf{lcl}\ \mathsf{E})\ \mathsf{This} = \mathsf{Some}\ (\mathsf{Class}\ \mathsf{i}\ \mathsf{C}) & \neg\ \mathsf{static\_method}\ \mathsf{E} \\ \mathsf{ic\_correct}\ (\mathsf{prg}\ \mathsf{E})\ \mathsf{i}\ \mathsf{C}' & \mathsf{super\_is\_accessible}\ (\mathsf{prg}\ \mathsf{E})\ \mathsf{C} \\ \multicolumn{2}{c}{\mathsf{check\_type}\ \mathsf{E}\ (\mathsf{Class}\ \mathsf{i}\ \mathsf{C}) = \mathsf{Some}\ (\mathsf{Class}\ \mathsf{i}\ \mathsf{C}')}\end{array}}{\mathsf{E} \models \mathsf{Super} := (\mathsf{Class}\ \mathsf{i}\ \mathsf{C}')}$$

```
constdefs
  super_is_accessible :: prog ⇒ tname ⇒ bool
  super_is_accessible G C ==
    (C ≠ Object) &
    (is_bclass G C ⟶ is_bclass G (super (the (bclass G C)))) &
    (is_tclass G C ⟶ (is_tclass G (super (the (tclass G C))) | super (the (tclass G C)) = Object)) &
    (is_rclass G C ⟶ (is_rclass G (super (the (rclass G C))) | is_bclass G (super (the (rclass G C)))))
```

Der Ausdruck TSuper wird analog zur Typisierung des Ausdrucks Super typisiert. Dies hat zur Folge, dass TSuper entgegen der intuitiven Annahme nicht auf den durch die implizite Superrollenklasse der den Typ von this bildenden Klasse gebildeten Typ typisiert wird. Stattdessen wird der durch die Konstante check_type bestimmte Typ zur Typisierung des Ausdrucks verwendet wird. Die Konstante check_type (siehe Abschnitt 6.3.3) bindet den der Variablen This im statischen lokalen Environment zugeordneten

Typ abhängig vom akutellen Teamkontext. Dadurch kann eine einheitliche Aussage über das Vorkommen von Rollentypen in einem Teamkontext formuliert werden (siehe Abschnitt 7.2).

$$\frac{\begin{array}{c}(\mathsf{lcl}\ \mathsf{E})\ \mathsf{This} = \mathsf{Some}\ (\mathsf{Class}\ \mathsf{TThis}\ \mathsf{C}) \qquad \neg\ \mathsf{static\_method}\ \mathsf{E} \\ \mathsf{ic\_correct}\ (\mathsf{prg}\ \mathsf{E})\ \mathsf{TThis}\ \mathsf{C}' \qquad \mathsf{tsuper\_is\_accessible}\ (\mathsf{prg}\ \mathsf{E})\ \mathsf{C} \\ \mathsf{check\_type}\ \mathsf{E}\ (\mathsf{Class}\ \mathsf{TThis}\ \mathsf{C}) = \mathsf{Some}\ (\mathsf{Class}\ \mathsf{TThis}\ \mathsf{C}')\end{array}}{\mathsf{E} \models \mathsf{TSuper} := (\mathsf{Class}\ \mathsf{TThis}\ \mathsf{C}')}$$

```
constdefs
  tsuper_is_accessible :: prog ⇒ tname ⇒ bool
  tsuper_is_accessible G C ==
    is_rclass G C & implicit (the (rclass G C)) ≠ None & is_rclass G (the (implicit (the (rclass G C))))
```

Der Ausdruck TargetExpr e wird auf den Typ des Ausdrucks e typisiert.

$$\frac{\mathsf{E} \models \mathsf{e} :- \mathsf{T}}{\mathsf{E} \models \mathsf{TargetExpr}\ \mathsf{e} := \mathsf{T}}$$

Das Lemma unique_wt bringt zum Ausdruck, dass die Typisierung von Termen relativ zu einem Environment E eindeutig ist. Es wird mittels *rule induction* über die Menge der wohlgetypten Terme *wt* nachgewiesen.

```
lemma unique_wt:
  E ⊨ t :: T
  ⟹ ∀ T'. E ⊨ t :: T' ⟶ T = T'
```

Das Lemma wt_Inj_elim bringt zum Ausdruck, dass Anweisungen der Typ PrimT Void, Ausdrücken vom Typ *expr* und *target* ein Typ und Listen von Ausdrücken eine Liste von Typen bei der Typisierung von Termen zugeordnet wird. Es wird mittels *rule induction* über die Menge der wohlgetypten Terme *wt* nachgewiesen und innerhalb des Nachweises einer Aussage bezüglich der Referenzierbarkeit von Rollenobjekten in Abschnitt 9.3 verwendet.

```
lemma wt_Inj_elim:
  E ⊨ t :: U ⟹
    (case t of
      Inl se ⇒ (case se of
                  In1 s ⇒ U=Inl (PrimT Void)
                | In2 e ⇒ (∃ T. U = Inl T)
                | In3 t ⇒ (∃ T. U = Inl T))
      | Inr el ⇒ (∃ T. U = Inr T))
```

7 Statische Analyse von ObjectTeams/Java

Die statische Analyse von ObjectTeams/Java besteht aus der Modellierung von Wohlgeformtheitsprädikaten. Diese stellen Überprüfungen dar, die beim Kompilieren von ObjectTeams/Java-Programmen durchgeführt werden. Diese Wohlgeformtheitsprädikate werden in Abschnitt 7.1 modelliert. Desweiteren wird eine Aussage über die Wohlkonstruiertheit von Typen formuliert, die bei der Typisierung von Termen relativ zu einem statischen Environment abgeleitet werden. Diese Aussage wird als statische Schichtenaussage bezeichnet, weil die Wohlkonstruiertheit eines Typs, der einen Rollentypen darstellt, unter anderem dessen Abhängigkeit von dem Kontext der Teamklasse - der Teamschicht - zum Ausdruck bringt, in der der Typ vorkommt. Die statische Schichtenaussage wird in Abschnitt 7.2 formuliert und nachgewiesen.

7.1 Modellierung der Wohlgeformtheit von ObjectTeams/Java-Programmen

Die Modellierung der Wohlgeformtheit von ObjectTeams/Java-Programmen besteht in Anlehnung an [KNO+02b] aus der Definition von Wohlgeformtheitsprädikaten für alle Bestandteile von Programmen - also Typ-, Attribut-, Methoden- und Klassendeklarationen. Desweiteren besteht sie aus der Definition eines Wohlgeformtheitsprädikats für ObjectTeams/Java-Programme zur Überprüfung der Wohlgeformtheit aller in einem Programm enthaltenen Klassendeklarationen relativ zu dem Kontext, in dem sie definiert sind.

7.1.1 Wohlgeformtheit von Typdeklarationen

Das Prädikat wf_ty_decl zur Überprüfung der Wohlgeformtheit von in ObjectTeams/Java-Programmen deklarierten Typen ist zusammen mit dem Prädikat wf_rty_decl jeweils als primitiv rekursive Funktion über die Typen *ty* und *ref_ty* modelliert.

Der Null-Typ darf nicht in Typdeklarationen vorkommen [GJS+00] (§4.1). Jeder primitive Typ ist wohlgeformt. Ein durch eine Klasse gebildeter Typ ClassT i C ist wohlgeformt, wenn folgendes gilt.

- Wenn der Typ ClassT i C in einer Basisklasse vorkommt, dann muss er durch eine Team- oder Basisklasse gebildet und um den Instanzkontext TheVoid angereichert sein.
- Wenn der Typ ClassT i C in einer Teamklasse vorkommt, dann darf es sich um einen durch eine Team- oder Basisklasse gebildeten und um den Instanzkontext TheVoid angereicherten Typ handeln, oder um einen durch eine Basisklasse gebildeten und um den Instanzkontext TThis angereicherten Typ, falls die Basisklasse eine Superklasse einer in der Teamklasse enthaltenen Rollenklasse darstellt, oder um einen durch eine in der Teamklasse enthaltene Rollenklasse gebildeten und um den Instanzkontext TThis angereicherten Typ.
- Wenn der Typ ClassT i C in einer Rollenklasse vorkommt, dann muss er dieselben Anforderungen erfüllen wie der Typ, der in einer Teamklasse vorkommt, bis auf dass die Rollenklasse beziehungsweise die Rollenklasse einer Superbasisklasse, durch die der Typ gebildet ist, in der die aktuelle Rollenklasse umschließenden Teamklasse enthalten sein muss.

```
consts
  wf_rty_decl :: prog ⇒ tname ⇒ ref_ty ⇒ bool
  wf_ty_decl :: prog ⇒ tname ⇒ ty ⇒ bool
primrec
  wf_rty_decl G actC NullT = False
  wf_rty_decl G actC (ClassT i C) =
    (if (is_bclass G actC)
    then (if ((is_tclass G C | is_bclass G C) & i = TheVoid) then True else False)
    else (if (is_tclass G actC)
        then (if ((is_tclass G C & i = TheVoid) |
                  (is_bclass G C & i = TheVoid) |
                  (is_bclass G C & i = TThis &
                    (∃ R ∈ set (roles (the (tclass G actC))). (R,C) ∈ (sub_class1 G)+)) |
                  (is_rclass G C & i = TThis & C ∈ set (roles (the (tclass G actC)))))
              then True else False)
         else (if (is_rclass G actC)
```

```
                then (if ((is_tclass G C & i = TheVoid) |
                          (is_bclass G C & i = TheVoid) |
                          (is_bclass G C & i = TThis &
                            (∃ R ∈ set (roles (the (tclass G (encl_class (the (rclass G actC)))))).
                              (R,C) ∈ (sub_class1 G)+)) |
                          (is_rclass G C & i = TThis &
                            C ∈ set (roles (the (tclass G (encl_class (the (rclass G actC))))))))
                      then True else False)
               else False)))

primrec
  wf_ty_decl G actC (PrimT pt) = True
  wf_ty_decl G actC (RefT rt) = wf_rty_decl G actC rt
```

7.1.2 Wohlgeformtheit von Attributdeklarationen

Eine Attributdeklaration ist wohlgeformt, wenn der deklarierte Typ des Attributs wohlgeformt ist.

```
constdefs
  wf_fdecl :: prog ⇒ tname ⇒ (vname * field) ⇒ bool
  wf_fdecl G actC f == wf_ty_decl G actC (field_ty (snd f))
```

7.1.3 Wohlgeformtheit von Methodendeklarationen

Die Wohlgeformtheit einer Methodendeklaration ist modelliert durch die Prädikate wf_mhead zur Überprüfung der Wohlgeformtheit des Methodenkopfes, wf_mdecl_wf zur Überprüfung von Einschränkungen bezüglich der gesamten Methodendeklaration, wf_mdecl_typing zur Überprüfung der Wohlgetyptheit des Methodenkörpers und des Methodenrückgabeausdrucks und den Prädikaten check_mdecls_ex und check_mdecls_ex_v zur Überprüfung von Einschränkungen der Methodendeklaration relativ zu dem Kontext, in dem sie definiert ist.

```
consts
  wf_mhead :: prog ⇒ tname ⇒ sig ⇒ methd ⇒ bool
  wf_mdecl_wf :: prog ⇒ tname ⇒ mdecl ⇒ bool
  create_inv_frame :: prog ⇒ tname ⇒ tname ⇒ sig ⇒ methd ⇒ inv_frame
  wf_mdecl_typing :: prog ⇒ tname ⇒ tname ⇒ mdecl ⇒ bool
  check_mdecls_ex :: prog ⇒ tname ⇒ 'a class_scheme ⇒ tname ⇒ bool
  check_mdecls_ex_v :: prog ⇒ tname ⇒ rclass ⇒ tname ⇒ bool
```

Ein Methodenkopf ist wohlgeformt, wenn die Anzahl der Parametertypen gleich der Anzahl der Parameternamen ist, alle Parameternamen verschieden sind, This nicht als expliziter Parametername in der Methodendeklaration enthalten ist, und alle Methodeneingabe- und der Methodenrückgabetyp wohlgeformt sind.

```
defs
  wf_mhead_def:
  wf_mhead G actC ==
    (λ sig m. length (param_tys sig) = length (params m) &
      distinct (params m) & This ∉ set (params m) &
      (∀ t ∈ set(param_tys sig). wf_ty_decl G actC t) &
      wf_ty_decl G actC (res_ty m))
```

Eine Methodendeklaration für sich betrachtet ist wohlgeformt, wenn ihr Methodenkopf wohlgeformt ist. Ist die Methode als abstrakt markiert, dann darf sie keinen Methodenkörper definieren, die Klasse, in der sie definiert ist, muss als abstrakt markiert sein, und es darf sich nicht um eine statische Methode handeln [GJS+00] (§8.4.3.2). Ist die Methode nicht als abstrakt markiert, dann muss sie einen Methodenkörper definieren, alle im Methodenkörper deklarierten lokalen Variablen müssen verschieden benannt und ihre Typen wohlgeformt sein, This darf nicht als explizite lokale Variable deklariert sein, und alle lokalen Variablen müssen hinsichtlich ihres Namens von den Methodenparameternamen verschieden sein.

```
defs
```

```
  wf_mdecl_wf_def:
  wf_mdecl_wf G actC ==
    (λ (sig,m). wf_mhead G actC sig m &
      (abstract m ⟶
        (body m = None & (abstract (the (class G actC))) & ¬ (static m))) &
      (¬(abstract m) ⟶ (body m ≠ None &
        (unique (lcl_vars (the (body m)))) &
        (∀ (vn,t) ∈ set(lcl_vars (the (body m))). wf_ty_decl G actC t) &
        This ∉ set (map fst (lcl_vars (the (body m)))) &
        (∀ pn ∈ set(params m). table_of (lcl_vars (the (body m))) pn = None))))
```

Die Definition der Konstante create_inv_frame dient zur Modellierung des Baus eines *invocation frame* beim Aufruf einer Methode. Es werden der Name der Klasse actC, in der die Methode definiert ist, der Name der Methode sig.name sig, die aufgerufen wird, alle in der Methodendefinition enthaltenen Typdeklarationen von lokalen Variablen und Methodenparametern, This bei Instanzmethoden und der Teamkontext bei Rollen- und Teammethoden gesetzt.

Alle Typen von Variablen des statischen lokalen Environments außer die This und TContext zugeordneten Typen werden mit Hilfe der Konstanten adjust_t relativ zu dem Teamkontext, zu dem die Methode typisiert wird, gebunden. Da die Konstante adjust_t als „partielle“ Funktion modelliert ist, wird Variablen des statischen lokalen Environments, für deren Typen die Konstante adjust_t kein definiertes Ergebnis liefert, None zugeordnet. Durch die Überprüfung der Wohlgeformtheit von Typdeklarationen von lokalen Variablen und von Parametern einer Methodendeklaration innerhalb der Prädikate wf_mdecl_wf und wf_mhead wird dies noch einmal separat formuliert. Wenn die in dieser Arbeit modellierten Wohlgeformtheitsüberprüfungen auf eine konkrete Implementierung abgebildet werden würden, dann bräuchte die Typisierbarkeit von Methodenanweisungen und des Methodenrückgabetyps einer Methodendeklaration erst gar nicht überprüft werden, wenn die Typdeklarationen von lokalen Variablen und Methodenparametern nicht wohlgeformt sind.

```
defs
  create_inv_frame_def:
  create_inv_frame G actC TC sig m ==
    inv_frame.make actC (sig.name sig)
      (λ vn. case vn of
        VName v ⇒ if (adjust_t G TC (the ((table_of (lcl_vars (the (body m)))
                                ((params m) [↦](param_tys sig))) vn)) = None)
                  then None
                  else ((table_of (lcl_vars (the (body m)))
                                ((params m) [↦](param_tys sig))) vn)
        | This ⇒ (if (¬(static m))
                 then (if (is_bclass G actC | is_tclass G actC)
                       then Some (Class TheVoid actC)
                       else Some (Class TThis actC))
                 else None)
        | TContext ⇒ (if (is_bclass G actC) then None else Some (Class TheVoid TC)))
```

Die Überprüfung der Wohlgetyptheit einer Methodendefinition ist durch das Prädikat wf_mdecl_typing modelliert. Die Typisierung der in dem Methodenkörper enthaltenen Anweisungen muss den Typ PrimT Void ergeben. Der bei der Typisierung des Methodenrückgabeausdrucks ermittelte Typ muss zum innerhalb der Methodendeklaration deklarierten Rückgabetyp beziehungsweise bei der Betrachtung der Wohlgeformtheit einer Methode entlang der Teamvererbungshierarchie zu dem relativ zum Teamkontext TC gebundenen, innerhalb der Methodendeklaration deklarierten Rückgabetyp weitbar sein.

```
defs
  wf_mdecl_typing_def:
  wf_mdecl_typing G actC TC ==
  (λ (sig,m).
    (G, create_inv_frame G actC TC sig m) |= stmt (the (body m)) :<> &
    (∃ RT. (G, create_inv_frame G actC TC sig m) |= res (the (body m)) :- RT &
      (RT, res_ty (the (adjust_m G TC m))) ∈ widen G))
```

Die Definition des Prädikats check_mdecls_ex dient zur Modellierung folgender Einschränkungen bei der Definition einer Methode im Kontext einer Klasse entlang der `extends`-Vererbungshierarchie.

- Wenn eine in der Klasse C definierte Methode eine gleichnamige aus einer direkten oder indirekten Superklasse geerbte Methode überschreibt, dann müssen die Parametertypen gleich und der Methodenrückgabetyp darf höchstens spezieller als der der überschriebenen Methode sein.
- Eine abstrakte Methode darf eine Instanzmethode oder eine abstrakte Methode überschreiben, jedoch keine statische Methode [GJS+00] (§8.4.3.1).
- Eine Klasse muss als abstrakt markiert sein, wenn sie eine abstrakte Methode erbt, aber nicht implementiert [GJS+00] (§8.1.1.1).

```
defs
  check_mdecls_ex_def:
  check_mdecls_ex G C c TC ==
    (∀ (s,m) ∈ set (methods c).
      (∀ D pTs' m'. cmethod_ex G (super c) TC (sig.name s) =
        Some (D, pTs', m') ⟶
          (pTs' ≠ None & m' ≠ None & adjust_s G TC (param_tys s) ≠ None &
          the (adjust_s G TC (param_tys s)) = the pTs' & adjust_m G TC m ≠ None &
          (res_ty (the (adjust_m G TC m)), res_ty (the m')) ∈ widen G) &
          ((abstract m) ⟶ ¬(static (the m')))) &
      (∀ D pTs' m' mn.
        cmethod_ex G (super c) TC mn = Some (D, pTs', m') ⟶
          (abstract (the m') ⟶
            ((∃ m''. (cmethod_ex G C TC mn = Some (C, pTs', m'') & ¬(abstract (the m'')))) |
            abstract c)))))
```

Die Definition des Prädikats check_mdecls_ex_v dient zur Modellierung der bei der Definition einer Rollenmethode einzuhaltenden Einschränkungen entlang der impliziten Vererbungshierarchie und hinsichtlich des Zusammenspiels des Erbens von Methoden entlang der impliziten und der `extends`-Vererbungshierarchie.

- Bei der Redefinition von Methoden entlang der impliziten Vererbungshierarchie dürfen die Methodenparametertypen nicht verändert werden bis auf die Bildung von Rollentypen abhängig vom aktuellen Teamkontext.

  Beispiel siehe Abbildung 5: Der Aufruf der Methode q innerhalb der Methode r der Rollenklasse BoardGame_Rule hat bei der Ausführung der Methode r im Kontext der Teamklasse TicTacToe zur Folge, dass dem Ausdruck NewC TThis BoardGame_Rule der Typ ClassT TThis TicTacToe_Rule zugewiesen wird. Das Attribut b wird abhängig vom statischen, relativ zum Teamkontext gebundenen Typ des Attributzugriffsausdrucks bestimmt. Bei der Typisierung der geerbten Methode r im Kontext der Teamklasse TicTacToe wird diesem der Typ int zugewiesen aufgrund der Redefinition der Methode q entlang der impliziten Vererbungshierarchie in der Rollenklasse TicTacTac_Rule inklusive einer Spezialisierung des Methodenrückgabetyps. Dies würde zu einem Typfehler führen.

- Erbt eine Rollenklasse eine abstrakte Methode, ohne sie zu implementieren, dann muss die Klasse als abstrakt markiert sein.

- Erbt eine Rollenklasse eine gleichnamige Methode entlang der impliziten und entlang der `extends`-Vererbungshierarchie, dann muss es eine Superrollenklasse geben, für die folgendes gilt. Erstens definiert sie die Methode, die die Rollenklasse entlang der impliziten und entlang der `extends`-Vererbungshierarchie erbt. Zweitens muss sie eine gemeinsame Superrollenklasse der Rollenklassen darstellen, von der die Rollenklasse die gleichnamige Methode erbt.

  Beispiel siehe Abbildung 5: Wird die Methode q der Rollenklasse TicTacToe_Rule zum Beispiel innerhalb der Methode p der Rollenklasse TicTacToe_SubRule aufgerufen, dann kommt es zu einem Typfehler durch das Erben der Methode n entlang der impliziten Vererbungshierarchie, weil die in der impliziten Superrollenklasse BoardGame_SubRule der Rollenklasse TicTacToe_SubRule die entlang der `extends`-Vererbungshierarchie geerbte in der Klasse TicTacToe_Rule definierte Methode n überschreibt.

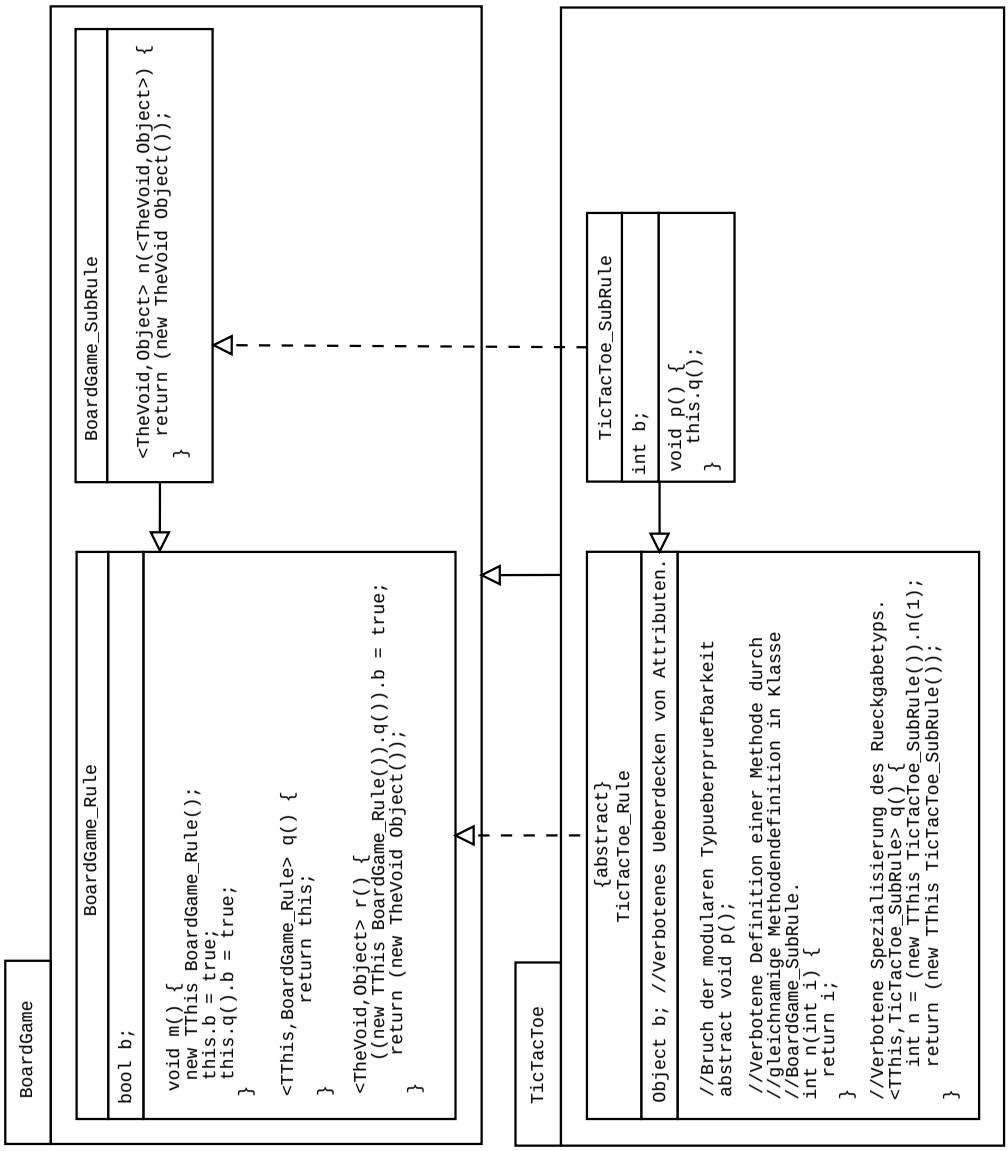

Abbildung 5: Fehlermöglichkeiten bei der Bildung von Subrollenklassen durch das Erben von Rollenklassen entlang der `extends`- und der impliziten Vererbungshierarchie.

```
defs
  check_mdecls_ex_v_def:
  check_mdecls_ex_v G C c TC ==
    (implicit c ≠ None ⟶
      (∀ (s,m) ∈ set (methods c).
        (∀ D pTs' m'. cmethod_v G (the (implicit c)) TC (sig.name s) =
          Some (D, pTs', m') ⟶
            (pTs' ≠ None & m' ≠ None & adjust_s G TC (param_tys s) ≠ None &
            the (adjust_s G TC (param_tys s)) = the pTs' & adjust_m G TC m ≠ None &
            res_ty (the (adjust_m G TC m)) = res_ty (the m')))) &
        (∀ D pTs' m' mn. cmethod_v G (the (implicit c)) TC mn =
          Some (D, pTs', m') ⟶
            (abstract (the m') ⟶
              ((∃ m''. (cmethod_v G C TC mn = Some (C, pTs', m'') & ¬(abstract (the m'')))) |
              abstract c))) &
        (∀ D_ex D_v pTs_ex pTs_v m_ex m_v mn.
            (cmethod_ex G (super c) TC mn = Some (D_ex, pTs_ex, m_ex) &
            cmethod_v G (the (implicit c)) TC mn =
              Some (D_v, pTs_v, m_v)) ⟶
              (∃ i_D_ex i_pTs_ex i_m_ex.
                cmethod_v G D_ex TC mn = Some (i_D_ex, i_pTs_ex, i_m_ex) &
                (D_v, i_D_ex) ∈ (sub_class1 G)*)))
```

7.1.4 Wohlgeformtheit von Klassendeklarationen

Die Wohlgeformtheit einer Klassendeklaration ist durch die Definition des Prädikats wf_class_decl für alle Klassenarten und durch die Definition der klassenspezifischen Prädikate wf_bclass_decl zur Modellierung basisklassenspezifischer Wohlgeformtheitsüberprüfungen, wf_tclass_decl zur Modellierung teamklassenspezifischer Wohlgeformtheitsüberprüfungen und wf_rclass_decl zur Modellierung rollenklassenspezifischer Wohlgeformtheitsüberprüfungen formalisiert.

```
consts
  wf_class_decl :: prog ⇒ tname ⇒ class ⇒ bool
  wf_bclass_decl :: prog ⇒ tname ⇒ class ⇒ tname ⇒ bool
  wf_tclass_decl :: prog ⇒ tname ⇒ tclass ⇒ tname ⇒ bool
  wf_rclass_decl :: prog ⇒ tname ⇒ rclass ⇒ tname ⇒ bool
```

Eine Klassendeklaration ist wohlgeformt, wenn alle innerhalb der Klasse definierten Attribute eindeutige Namen haben und wohlgeformt sind, und alle innerhalb der Klasse definierten Methoden wohlgeformt sind und verschiedene Namen haben.

Das Prädikt wf_class_decl wurde nach der Formalisierung eines Typsystems für ObjectTeams/Java mit dem Ziel definiert, alle unabhängig vom dem Kontext, in dem eine Klasse definiert ist, überprüfbaren Einschränkungen, die für alle drei Klassenarten zu überprüfen sind, innerhalb der Definition eines Prädikats zu kapseln. In Abschnitt 7.1.5 werden Überlegungen dargelegt, ob die Wohlgeformtheit einer Klassendeklaration in dem Kontext, in dem sie definiert ist, die Wohlgeformtheit dieser Klassendeklaration in allen Kontexten, in denen sie verfeinert und redefiniert werden kann, impliziert. Wenn dem so ist, dann sollten die Wohlgeformtheitsüberprüfungen von Attribut- und Methodendeklarationen auch um den Teamkontext, relativ zu dem ihre Wohlgeformtheit überprüft beziehungsweise nachgewiesen wird, parametrisiert werden.

```
defs
  wf_class_decl_def:
  wf_class_decl G C c ==
    unique (attrs c) &
    (∀ f ∈ set (attrs c). wf_fdecl G C f) &
    (∀ m ∈ set (methods c). wf_mdecl_wf G C m &
      distinct (map (λ(s,m). (sig.name s)) (methods c)))
```

Eine Basisklassendeklaration ist wohlgeformt, wenn die Methodenkörper und Methodenrückgabeaus-

drücke aller in ihr definierten, nicht abstrakten Methoden, wohlgetypt sind, und, wenn es sich nicht um die Klasse Object handelt, alle in ihr definierten Methoden im Kontext der Klasse wohlgeformt sind. Dies wird durch Anwendung des Prädikats check_mdecls_ex modelliert.

```
defs
  wf_bclass_decl_def:
  wf_bclass_decl G C c TC ==
    (∀ m ∈ set (methods c). ¬(abstract (snd m)) ⟶ wf_mdecl_typing G C TC m) &
    (C ∉ Object ⟶ check_mdecls_ex G C c TC)
```

Eine Teamklassendeklaration ist wohlgeformt, wenn keines der in ihr definierten Attribute mit einem um den Instanzkontext TThis angereicherten Typ überdeckt wird, sie keine statischen Methoden definiert, die Methodenkörper und Methodenrückgabeausdrücke aller in ihr definierten, nicht abstrakten Methoden wohlgetypt sind, und, wenn sie nicht die Klasse Team darstellt, alle in ihr definierten Methoden im Kontext der Klasse wohlgeformt sind - modelliert durch Anwendung des Prädikats check_mdecls_ex - und sie als abstrakt markiert ist, wenn die speziellste Rollenklasse einer Rollenklassenhierarchie entlang der `extends`-Vererbungshierarchie innerhalb dieser Teamklasse abstrakt ist [Her03c] (§7.1(c)).

```
defs
  wf_tclass_decl_def:
  wf_tclass_decl G C c TC ==
  unique (filter (λ(fn,fd,f). ic (the (field_ty f)) ≠ None & the (ic (the (field_ty f))) = TThis)
    (map (λ((fn,fd),f). (fn,(fd,f))) (cfields_ex G C TC))) &
  (∀ m ∈ set (methods c). ¬(static (snd m)) &
      ¬(abstract (snd m)) ⟶ wf_mdecl_typing G C TC m) &
  (C ≠ Team ⟶ check_mdecls_ex G C (class.truncate c) TC) &
    (∃ R ∈ set (roles c).
      (abstract (the (rclass G R)) &
      ¬(∃ R' ∈ set (roles c). (R',R) ∈ (sub_class1 G)^+ & ¬(abstract (the (rclass G R')))))) ⟶ abstract c)
```

Eine Rollenklassendeklaration ist wohlgeformt, wenn sie keine statischen Methoden definiert, die Methodenkörper und Methodenrückgabeausdrücke aller in ihr definierten, nicht abstrakten Methoden wohlgetypt sind, alle entlang der impliziten Vererbungshierarchie geerbten Attribute einen eindeutigen Namen haben, also entlang der impliziten Vererbungshierarchie keine Attribute überdeckt werden, alle in ihr definierten Methoden im Kontext der `extends`-Vererbungshierarchie wohlgeformt sind, und, falls sie eine implizite Superrollenklasse besitzt, alle in ihr definierten Methoden im Kontext der impliziten und im Zusammenspiel der impliziten und extends-Vererbungshierarchie wohlgeformt sind.

Attribute dürfen entlang der impliziten Vererbungshierarchie nicht überdeckt werden, weil es sonst zum Beispiel bei Ausführung der Methode m der Rollenklasse BoardGame_Rule im Kontext der Teamklasse TicTacToe durch Überdecken des Attributs b zu einem Typfehler während der Zuweisung des Literals `true` an das Attribut b kommen würde (siehe Abbildung 5).

```
defs
  wf_rclass_decl_def:
  wf_rclass_decl G C c TC ==
  (∀ m ∈ set (methods c). ¬(static (snd m)) &
      ¬(abstract (snd m)) ⟶ wf_mdecl_typing G C TC m) &
  unique (map (λ((fn,fd),f). (fn,(fd,f))) (cfields_v G C TC)) &
  check_mdecls_ex G C c TC & (implicit c ≠ None ⟶ check_mdecls_ex_v G C c TC)
```

7.1.5 Wohlgeformtheit von Programmen

Die Wohlgeformtheit eines Programms ist durch das Prädikat wf_prog definiert. Es modelliert die Überprüfung, ob das Programm wohlstrukturiert ist, und alle in einem Programm enthaltenen Klassendeklarationen wohlgeformt sind.

```
constdefs
  wf_prog :: prog ⇒ bool
  wf_prog G ==
  ws_prog G &
```

```
unique (attrs c) &
(∀ c ∈ set (bclass_decls G
  (map (λ(C,c). (C, class.truncate c)) (tclass_decls G))
  (map (λ(C,c). (C, class.truncate c)) (rclass_decls G))).
    wf_class_decl G (fst c) (snd c)) &
(∀ b ∈ set (bclass_decls G). wf_bclass_decl G (fst b) (snd b) (fst b)) &
(∀ t ∈ set (tclass_decls G). wf_tclass_decl G (fst t) (snd t) (fst t)) &
(∀ r ∈ set (rclass_decls G).
  wf_rclass_decl G (fst r) (snd r) (encl_class (snd r)))
```

Innerhalb des Prädikats wf_prog G zur Überprüfung der Wohlgeformtheit eines Programms G wird die Wohlgeformtheit aller in diesem Programm enthaltenen Klassendeklarationen relativ zu dem Kontext, in dem die Klassen definiert sind, überprüft. Bei der Verfeinerung von Teamklassen werden innerhalb allgemeinerer Teamkontexte definierte Attribut- und Methodendeklarationen von Team- und Rollenklassen vererbt. Durch die Redefinition von Rollenklassen bei der Bildung von Subteamklassen werden diese jedoch nicht unverändert in spezielleren Teamkontexten wiederverwendet. Stattdessen werden in Attribut- und Methodendeklarationen enthaltene Rollentypen abhängig von dem Teamkontext, in dem sie wiederverwendet werden, gebunden. Dies hat zur Folge, dass die Wohlgeformtheit einer Klassendeklaration nicht nur in dem Kontext, in dem sie definiert ist, betrachtet werden muss, sondern in allen Kontexten entlang der entsprechenden Teamvererbungshierarchie, in denen Eigenschaften von ihr geerbt werden. Folgende Überlegungen sind im Rahmen dieser Arbeit dazu erbracht.

1. Das im folgenden beschriebene und in Abbildung 5 dargestellte Beispiel bringt zum Ausdruck, dass die Wohlgeformtheitsüberprüfung einer Klasse bei Überprüfung der Wohlgeformtheit aller in der Klasse definierten Eigenschaften durch das Erben von Eigenschaften durch die derzeitigen Regeln, wann eine Teamklasse als abstrakt markiert werden muss, gebrochen wird.

   Das Erben der in der Rollenklasse BoardGame_Rule der Teamklasse BoardGame definierten Methode m durch die Rollenklasse TicTacToe_Rule der Subteamklasse TicTacToe, die unter anderem eine Instanz der Rollenklasse BoardGame_Rule erzeugt, hat zur Folge, dass die Teamklasse TicTacToe als abstrakt markiert werden muss, weil in ihrem Kontext die Rollenklasse TicTacToe_Rule als abstrakt redefiniert ist.

   Es ist eine offene Fragestellung, ob weitere Einschränkungen formuliert werden können, um das obige Szenario auszuschließen und zu vermeiden, dass bei der Prüfung der Wohlgeformtheit einer Klasse auch die Überprüfung der Wohlgeformtheit aller geerbten Eigenschaften miteinbezogen werden muss.

2. Abgesehen von dem unter erstens beschriebenen Szenario sollte die Wohlgeformtheit einer Klassendeklaration in dem Kontext, in dem sie definiert ist, die Wohlgeformtheit dieser in verfeinerten Kontexten implizieren.

   Um dies zu zeigen muss erstens nachgewiesen werden, dass die Wohlgetyptheit der in einem Methodenkörper enthaltenen Methodenanweisungen und des Methodenrückgabeausdrucks die Wohlgetyptheit dieser in verfeinerten Kontexten impliziert. Dies bedeutet, dass für jede Typisierungsregel nachgewiesen werden muss, dass die Wohlgetyptheit eines Terms relativ zu einem statischen Environment dessen Wohlgetyptheit relativ zu diesem um einen verfeinerten Teamkontext aktualisierten statischen Environment impliziert. Dabei muss die Gültigkeit von in den Typisierungsregeln formulierten Einschränkungen relativ zum aktualisierten statischen Environment nachgewiesen und eine Aussage über die Bildbarkeit von Rollentypen relativ zum verfeinerten Teamkontext getroffen werden.

   Zweitens muss die Gültigkeit aller Einschränkungen, die für eine wohlgeformte Klassendeklaration gelten, im verfeinerten Teamkontext nachgewiesen werden. Dies gilt für Eigenschaften, die unabhängig von dem Teamkontext, relativ zu dem die Wohlgeformtheit einer Klassendeklaration überprüft wird, definiert sind, per Voraussetzung. Für Eigenschaften, deren Gültigkeit abhängig von einem Teamkontext überprüft wird, sollte dies gezeigt werden können, weil die Verfeinerung des Teamkontextes „nur“ die Bindung von Rollentypen beeinflußt.

7.2 Nachweis einer statischen Schichtenaussage

Es wird eine Aussage über die bei der Typisierung von Termen relativ zu einem statischen Environment abgeleiteten Typen formuliert und nachgewiesen. Sie bringt zum Ausdruck, dass bei der Typisierung von Termen aus einem ObjectTeams/Java-Programm wohlkonstruierte Typen abgeleitet werden, wenn dieses ObjectTeams/Java-Programmm wohlgeformt ist. Die Wohlkonstruiertheit von Typen ist im folgenden unter anderem durch das Prädikat wc_type definiert. Es beinhaltet, dass bei der Typisierung von Termen abgeleitete Rollentypen durch Rollenklassen oder Superbasisklassen von Rollenklassen gebildet sind, die im Teamkontext des statischen Environments enthalten sind, relativ zu dem die Terme typisiert werden. Die Aussage wird als statische Schichtenaussage bezeichnet. Sie ist in Abbildung 4 durch Auffassung einer Teamklasse samt ihrer Rollenklassen als Teamschicht visualisiert. Es wird gezeigt, dass in einer Teamschicht nur Rollentypen vorkommen, die durch Rollenklassen oder Superbasisklassen dieser Teamschicht gebildet sind. Dies gilt auch für die Typen von Termen, die in allgemeineren Teamschichten definiert sind. Um dies zu visualisieren, sind in Abbildung 4 die in einer impliziten Vererbungsbeziehung stehenden Rollenklassen zusammengefasst.

Die statische Schichtenaussage wird in den Abschnitten 9.2 und 9.3 verwendet, um eine Aussage über die Typzuverlässigkeit des in Kapitel 6 modellierten Typsystems für ObjectTeams/Java zu betrachten, und eine Aussage über das *Confinement* von Rollenobjekten zu analysieren.

Die Wohlkonstruiertheit von Typen ist durch die Prädikate wc_type, is_wc_rtype und is_wc_type beschrieben, wobei letztere als primitiv rekursive Funktionen über die Typen ref_ty und ty formalisiert sind.

```
consts
  is_wc_rtype :: env ⇒ ref_ty ⇒ bool
  is_wc_type :: env ⇒ ty ⇒ bool
  wc_type :: env ⇒ tys ⇒ bool
```

Der Null-Typ und jeder primitive Typ sind wohlkonstruiert. Ein Klassentyp ist wohlkonstruiert, wenn folgendes gilt.

- Wenn ein Typ durch den Instanzkontext TheVoid und eine Basis- oder Teamklasse gebildet ist, dann ist er wohlkonstruiert.
- Wenn ein Typ durch eine Basisklasse und den Instanzkontext TThis gebildet ist, der Teamkontext innerhalb des statischen Environments korrekt gesetzt ist, und die Basisklasse eine Superklasse einer im Teamkontext enthaltenen Rollenklasse darstellt, dann ist er wohlkonstruiert.
- Wenn ein Typ durch eine Rollenklasse und den Instanzkontext TThis gebildet ist, der Teamkontext innerhalb des statischen Environments korrekt gesetzt ist, und die Rollenklasse im Teamkontext enthalten ist, dann ist er wohlkonstruiert.

```
primrec
  is_wc_type_NT: is_wc_rtype E (NullT) = True
  is_wc_type_Class:
  is_wc_rtype E (ClassT i C) =
    (((is_bclass (prg E) C | is_tclass (prg E) C) & i = TheVoid) |
    (is_bclass (prg E) C & i = TThis & check_tcontext E ≠ None & roles_tcontext E ≠ None &
      (∃ R ∈ set (the (roles_tcontext E)). (R,C) ∈ (sub_class1 (prg E))+)) |
    (is_rclass (prg E) C & i = TThis & check_tcontext E ≠ None &
      roles_tcontext E ≠ None & C ∈ set (the (roles_tcontext E))))
primrec
  is_wc_type_PrimT: is_wc_type E (PrimT pt) = True
  is_wc_type_RefT: is_wc_type E (RefT rt) = is_wc_rtype E rt
defs
  wc_type_def:
  wc_type E T ==
  (case T of
    Inl t ⇒ is_wc_type E t
    | Inr ts ⇒ (∀ t ∈ set ts. list_all (is_wc_type E) ts))
```

Die statische Schichtenaussage layers_of_static_role_types bringt zum Ausdruck, dass bei der Typisierung eines beliebigen Terms t relativ zu einem beliebigen, aber fixen statischen Environment E unter der Voraussetung, dass das im statischen Environment enthaltene Programm wohlgeformt ist, ein statischer Typ T abgeleitet wird, der wohlkonstruiert ist.

```
theorem layers_of_static_role_types [rule_format]:
  E |= t :: T
  ⟹ wf_prog (prg E) —→ wc_type E T
```

Die statische Schichtenaussage layers_of_static_role_types ist mittels *rule induction* über die Menge der wohltypisierten Terme wt nachgewiesen. Die einzelnen Beweissubziele werden wie folgt nachgewiesen.

Die bei der Typisierung der Terme Skip, Expr, Comp, Cond, Loop und Inst abgeleiteten primitiven Typen sind per Definition wohlkonstruiert (siehe is_wc_type_PrimT).

Die Wohlkonstruiertheit der bei der Typisierung der Terme LAss, FAss und TargetExpr abgeleiteten Typen wird unter Verwendung der Induktionsvoraussetzung nachgewiesen.

Die Wohlkonstruiertheit der bei der Typisierung der Terme NewC, Super und TSuper abgeleiteten Typen wird durch Anwendung des Lemmas check_type_is_wc_Class nachgewiesen. Das Lemma check_type_is_wc_Class bringt zum Ausdruck, dass der durch die Konstante check_type bestimmte Typ wohlkonstruiert ist. Das Lemma check_type_is_wc_Class ist mittels Fallunterscheidung über den Instanzkontext eines Typs und unter Verwendung des Lemmas rrc_rclass_in_roles_tcontext nachgewiesen. Das Lemma rrc_rclass_in_roles_tcontext bringt zum Ausdruck, dass die durch die Konstante rrc bestimmte Klasse eine Rollenklasse ist, und dass diese Rollenklasse eine im aktuellen Teamkontext sichtbare ist.

```
lemma check_type_is_wc_Class:
  check_type E (Class i C) = Some (Class i C')
  ⟹ is_wc_type E (Class i C')
```

```
lemma rrc_rclass_in_roles_tcontext:
  [|rrc (prg E) (the (check_tcontext E)) C ≠ None;
  C' = the (rrc (prg E) (the (check_tcontext E)) C);
  is_rclass (prg E) C;
  check_tcontext E ≠ None; roles_tcontext E ≠ None|]
  ⟹ is_rclass (prg E) C' & C' ∈ set (the (roles_tcontext E))
```

Die Wohlkonstruiertheit der bei der Typisierung der Terme Cast und LAcc abgeleiteten Typen ist mit Hilfe der beiden Lemmata check_type_is_wc_T_Class und check_local_type_is_wc_T_Class nachgewiesen. Sie bringen zum Ausdruck, dass die durch die Konstanten check_type und check_local_type bestimmten Typen wohlkonstruiert sind.

```
lemma check_type_is_wc_T_Class:
  check_type E T = Some (Class i C')
  ⟹ is_wc_type E (Class i C')
```

```
lemma check_local_type_is_wc_T_Class:
  check_local_type E T = Some (Class i C')
  ⟹ is_wc_type E (Class i C')
```

Zum Nachweis der Wohlkonstruiertheit des bei der Typisierung des Terms Tthis abgeleiteten Typs wird das Lemma type_Tthis_is_wc_type verwendet. Es bringt zum Ausdruck, dass der durch die Konstante type_Tthis bestimmte Typ wohlkonstruiert ist. Zum Nachweis des Lemmas type_Tthis_is_wc_type wird das Lemma encl_class_is_tclass verwendet.

```
lemma type_Tthis_is_wc_type:
  [|type_Tthis E = Some T; wf_prog (prg E)|]
  ⟹ is_wc_type E T
```

```
lemma encl_class_is_tclass:
  [|C' = encl_class (the (rclass G C)); ws_prog G; is_rclass G C|]
  ⟹ is_tclass G C'
```

Zum Nachweis der Wohlkonstruierheit der bei der Typisierung der Terme FAcc und Call abgeleiteten Typen werden die Lemmata field_ty_is_wc_type und res_ty_is_wc_type verwendet. Sie bringen zum

Ausdruck, dass die durch die Konstanten cfield und cmethod bestimmten Attribut- beziehungsweise Methodenrückgabetypen wohlkonstruiert sind. Beim Nachweis dieser Lemmata wird das Lemma adjust_t_is_wc_type verwendet. Das Lemma adjust_t_is_wc_type bringt zum Ausdruck, dass die Konstante adjust_t einen wohlkonstruierten Typ bestimmt. Die Konstante adjust_t wird beim Zugriff auf ein Attribut und beim Aufruf einer Methode angewendet, um den Attributtyp und die Methodenparametertypen abhängig vom akutellen Teamkontext zu binden. Für den Nachweis der Lemmata field_ty_is_wc_type und res_ty_is_wc_type werden die Lemmata cfields_ex_class_is_wc_type und cfields_v_rclass_is_wc_type beziehungsweise cmethods_ex_class_is_wc_type und cmethods_v_rclass_is_wc_type verwendet. Diese sind mittels Induktion über die `extends`- und implizite Vererbungshierarchie nachgewiesen. Innerhalb dieser Beweise werden die Lemmata adjust_m_is_wc_type und adjust_t_is_wc_type verwendet.

```
lemma field_ty_is_wc_type:
  [|cfield (prg E) C (the (check_tcontext E)) fn = Some (fd, Some f);
  field_ty f = Class i C'; check_tcontext E = Some y; wf_prog (prg E)|]
  ⟹ is_wc_type E (field_ty f)

lemma res_ty_is_wc_type:
  [|cmethod (prg E) C' (the (check_tcontext E)) mn =
  Some (md, Some pTs', Some m);
  res_ty m = Class i C; check_tcontext E = Some y; wf_prog (prg E)|]
  ⟹ is_wc_type E (res_ty m)

lemma adjust_t_is_wc_type:
  [|wf_prog (prg E); check_tcontext E = Some y; adjust_t (prg E) y T = Some yaa|]
  ⟹ is_wc_type E yaa

lemma cfields_ex_class_is_wc_type:
  [|is_class (prg E) C; wf_prog (prg E); check_tcontext E = Some y|]
  ⟹ ∀ f ∈ set (cfields_ex (prg E) C y). snd f ≠ None ⟶
       is_wc_type E (field_ty (the (snd f)))

lemma cfields_v_rclass_is_wc_type:
  [|is_rclass (prg E) C; wf_prog (prg E);
  check_tcontext E = Some y; is_tclass (prg E) y|]
  ⟹ ∀ f ∈ set (cfields_v (prg E) C y). snd f ≠ None ⟶
       is_wc_type E (field_ty (the (snd f)))

lemma cmethods_ex_class_is_wc_type:
  [|is_class (prg E) C; wf_prog (prg E);
  check_tcontext E = Some y|]
  ⟹ ∀ (mn,md,pTs,m) ∈ set (cmethods_ex (prg E) C y). m ≠ None ⟶
       is_wc_type E (res_ty (the m))

lemma cmethods_v_rclass_is_wc_type:
  [|is_rclass (prg E) C; wf_prog (prg E); check_tcontext E = Some y|]
  ⟹ ∀ (mn,md,pTs,m) ∈ set (cmethods_v (prg E) C y). m ≠ None ⟶
       is_wc_type E (res_ty (the m))

lemma adjust_m_is_wc_type:
  [|wf_prog (prg E); Some ya = adjust_m (prg E) y bb; check_tcontext E = Some y|]
  ⟹ is_wc_type E (res_ty ya)
```

7.2.1 Aus der statischen Schichtenaussage ableitbare Aussagen

Aus der statischen Schichtenaussage layers_of_static_role_types werden die beiden Lemmata role_type_tcontext_exists und occurence_of_role_types abgeleitet. Sie bringen folgendes zum Ausdruck.

- Lemma role_type_tcontext_exists: Wenn für einen beliebigen Term t relativ zu einem statischen Environment E ein Rollentyp abgeleitet wird, und das im statischen Environment enthaltene Programm wohlgeformt ist, dann existiert auch ein Teamkontext im statischen Environment.
- Lemma occurence_of_role_types: Wenn für einen beliebigen Term t relativ zu einem statischen Envi-

ronment E ein Rollentyp abgeleitet wird, und das im statischen Environment enthaltene Programm wohlgeformt ist, dann tritt dieser Term in einer Team- oder Rollenklasse, und nicht in einer Basisklasse auf.

```
lemma role_type_tcontext_exists:
  [| E |= t :: T; wf_prog (prg E); role_type (prg E) T|]
  ⟹ tcontext_exists E

lemma occurence_of_role_types:
  [| E |= t :: T; wf_prog (prg E); role_type (prg E) T|]
  ⟹ ((is_tclass (prg E) (act_cls E) | is_rclass (prg E) (act_cls E)) &
      ¬(is_bclass (prg E) (act_cls E)))
```

Im folgenden werden die Prädikate tcontext_exists und role_type erläutert, die innerhalb der beiden obigen Lemmata verwendet werden.

Das Prädikat tcontext_exists modelliert die Überprüfung, ob der Teamkontext des statischen Environments korrekt gesetzt ist.

```
constdefs
  tcontext_exists :: env ⇒ bool
  tcontext_exists E == check_tcontext E ≠ None & roles_tcontext E ≠ None
```

Das Prädikat role_type überprüft für einen Typ und eine Liste von Typen, ob der Typ oder ein Typ aus der Liste von Typen einen Rollentyp darstellt. Diese Überprüfung ist durch Anwendung der Prädikate is_role_rtype und is_role_type modelliert. Sie prüfen, ob ein Typ einen Rollentyp darstellt, das heisst, ob der Typ durch den Instanzkontext TThis und eine Rollen- oder Basisklasse gebildet ist. Sie sind als primitiv rekursive Funktionen über die Typen *ref_ty* und *ty* modelliert.

```
consts
  is_role_rtype :: prog ⇒ ref_ty ⇒ bool
  is_role_type :: prog ⇒ ty ⇒ bool
  role_type :: prog ⇒ tys ⇒ bool

primrec
  is_role_type_NT:
  is_role_rtype G (NullT) = False
  is_role_type_Class:
  is_role_rtype G (ClassT i C) = ((is_rclass G C | is_bclass G C) & i = TThis)

primrec
  is_role_type_PrimT:
  is_role_type G (PrimT pt) = False
  is_role_type_Class:
  is_role_type G (RefT rt) = is_role_rtype G rt

defs
  role_type_def:
  role_type G T ==
  (case T of
    Inl t ⇒ is_role_type G t
    | Inr ts ⇒ (∃ t ∈ set ts. is_role_type G t))
```

8 Modellierung einer Semantik für ObjectTeams/Java

Eine Semantik für ObjectTeams/Java ist durch die Modellierung eines Programmzustands in Abschnitt 8.1 und die Definition von Regeln zur Auswertung von Termen in Abschnitt 8.2 formalisiert. Beide Modellierungen sind stark an die in [KNO+02a] enthaltene Formalisierung einer Semantik für Java angelehnt.

8.1 Modellierung eines Programmzustands

In Abschnitt 8.1.1 wird die Modellierung von Objekten vorgestellt. Die Formalisierung eines dynamischen Aufrufrahmens, eines Heaps und eines erweiterten Programmzustands ist in Abschnitt 8.1.2 enthalten.

8.1.1 Modellierung von Objekten

Ein Objekt - formalisiert durch den Typ *obj* - ist gekennzeichnet durch einen *tag* und speichert die Werte seiner Attribute vom Typ *val* in einem *table*. Bei dem Typ der Schlüsselelemente dieses *table* handelt es sich um den Namen eines Attributs kombiniert mit dem Namen der Klasse, in der das Attribut definiert ist, um gleichnamige entlang der `extends`-Vererbungshierarchie geerbte Attribute unterscheiden zu können.

Das Typsynonym *tag* modelliert Informationen, durch die Objekte ausgezeichnet sind. Dabei handelt es sich um den Namen der Klasse, von der das Objekt eine Instanz darstellt, und die Referenz auf das Objekt, in dem es enthalten ist - gegebenenfalls in keinem.

```
types obj_tag = dyn_ic * tname

record obj =
    tag :: obj_tag
    values :: ((vname * tname), val)table
```

Folgende Konstanten werden zum Zugriff auf den Namen der Klasse, von der das Objekt eine Instanz darstellt (obj_class), zum Zugriff auf den dynamischen Typ eines Objekts (obj_ty), zum Zugriff auf den dynamsichen Instanzkontext eines Objekts (obj_dic) und zur Überprüfung, ob ein Objekt einen korrekten Typ relativ zum Programm G besitzt (is_obj_ty), modelliert.

```
constdefs
    obj_class :: obj ⇒ tname
    obj_class obj == snd (tag obj)

    obj_ty :: obj ⇒ dyn_ty
    obj_ty obj == DynClass (fst (tag obj)) (snd (tag obj))

    obj_dic :: obj ⇒ dyn_ic
    obj_dic obj == fst (tag obj)

    is_obj_ty :: prog ⇒ obj ⇒ bool
    is_obj_ty G obj ==
    (case (obj_dic obj) of
      DynTheVoid ⇒ (is_bclass G (obj_class obj) | is_tclass G (obj_class obj))
      | DynTId l ⇒ (is_rclass G (obj_class obj)))
```

Die Konstante upd_obj modelliert die Aktualisierung eines Attributwertes eines Objekts. Die Konstante var_tys dient zur Bestimmung aller Attribute eines Objekts angereichert um den Namen der Klasse, in der das Attribut definiert ist, und des Typs, der ihm bei der Deklaration zugewiesen ist. Die Konstante init_vals modelliert die Default-Initialisierung aller Attribute eines Objekts [GJS+00](§4.5.5) Diese ist innerhalb der Konstante default_val modelliert, die mittels primitiver Rekursion über den Typ *ty* definiert ist und primitive Typen deren entsprechenden Wert und Referenztypen den Null-Wert zuordnet [KNO+02a] (Value.thy).

```
constdefs
    upd_obj :: obj ⇒ tname ⇒ vname ⇒ val ⇒ obj
    upd_obj obj T fn v ==
      obj (|values := (values obj)((fn,T) ↦v) |)
```

```
var_tys :: prog ⇒ tname ⇒ tname ⇒ ((vname * tname),ty)table
var_tys G C TC ==
  option_map (field_ty o the) o
    (if (is_rclass G C)
    then (table_of (cfields_ex G C TC)) ++ (table_of (cfields_v G C TC))
    else table_of (cfields_ex G C TC))

init_vals :: ('a,ty)table ⇒ ('a,val)table
init_vals vs == option_map default_val o vs
```

8.1.2 Modellierung eines dynamischen Aufrufrahmens und eines Heaps

Der Typ *dyn_inv_frame* ist zur Formalisierung des dynamischen Aufrufrahmens beim Aufruf einer Methode in Analogie zum Typ *inv_frame* (siehe Abschnitt 6.3.1) modelliert. Innerhalb des dynamischen lokalen Environments, das durch das Typsynonym *locals* modelliert ist, werden jedoch Variablen nicht Typen zugeordnet, sondern Werte.

```
types locals = (vname,val)table

record dyn_inv_frame =
    dyn_act_cls :: tname
    dyn_act_method :: mname
    dyn_act_mode :: inv_mode
    dyn_lenv :: locals
```

Sich bei der Ausführung eines Programms auf dem Heap befindende Objekte werden durch Verweise vom Typ *loc* (siehe Abschnitt 6.1.4) referenziert. Der Zustand eines Programms setzt sich aus einem Heap - modelliert durch den Typ *aheap* - und einem dynamischen Aufrufrahmen zusammen und wird durch das Typsynonym *state* formalisiert. Das Typsynonym *xstate* dient zur Modellierung eines um eine Exception-Komponente erweiterten Zustands zur Realisierung des Exception-Konzeptes von Java.

```
types
    aheap = (loc, obj)table
    state = aheap * dyn_inv_frame
    xstate = xcpt option * state
```

Die Konstanten heap, locals, Norm modellieren den Zugriff auf den Heap eines Programmzustandes, das dynamische lokale Environment eines Programmzustands und den Zugriff auf einen Programmzustand ohne Exception-Komponente.

```
constdefs
    heap :: state ⇒ aheap
    heap s == fst s

    locals :: state ⇒ locals
    locals s == dyn_lenv (snd s)

    Norm :: state ⇒ xstate
    Norm s == (None,s)
```

Die Konstanten val_this, val_TContext, id_TContext und dyn_cls_TContext modellieren den Zugriff auf den Wert von This, auf den Wert von TContext, auf die Referenz auf das durch die Variable TContext referenzierte Teamobjekt, und auf dessen Klasse, von der es eine Instanz ist.

```
constdefs
    val_this :: state ⇒ val option
    val_this s == (locals s) This

    val_TContext :: state ⇒ val option
    val_TContext s == (locals s) TContext

    id_TContext :: state ⇒ loc
    id_TContext s == the_Addr (the ((locals s) TContext))
```

dyn_cls_TContext :: *state* ⇒ *tname*
dyn_cls_TContext s == obj_class (the ((heap s) (id_TContext s)))

Die Definitionen der Konstanten upd_locals, c_hupd und new_Addr formalisieren die Aktualisierung des dynamischen lokalen Environments, die Aktualisierung des Heaps und die Erzeugung einer neuen Referenz bei der Instanziierung einer Klasse. Die Konstanten c_hupd und new_Addr und Aussagen über diese sind aus [KNO+02a] übernommen.

```
constdefs
```
upd_locals :: *state* ⇒ *vname* ⇒ *val* ⇒ *state*
upd_locals s va v == (heap s, (snd s) (|dyn_lenv := (locals s) (va ↦v) |))

c_hupd :: *aheap* ⇒ *xstate* ⇒ *xstate*
c_hupd h'== (%(x,(h,l)). if (x = None) then (None,(h',l)) else (x,(h,l)))

new_Addr :: *aheap* ⇒ loc ∗ *xcpt option*
new_Addr h ==
  SOME (a,x). (h a = None & x = None) | x = Some OutOfMemory

Die Konstanten raise_if und np modellieren das Werfen von Exceptions. Ihre Definitionen und Aussagen über sie sind aus [KNO+02a] übernommen.

```
constdefs
```
raise_if :: *bool* ⇒ *xcpt* ⇒ *xcpt option* ⇒ *xcpt option*
raise_if c x xo == if (c & xo = None) then Some x else xo

np :: *val* ⇒ *xcpt option* ⇒ *xcpt option*
np v == raise_if (v = Null) NullPointer

8.2 Modellierung von Auswertungsregeln

Es wird eine operationale Semantik für Terme von ObjectTeams/Java-Programmen definiert. Bei der Auswertung eines Terms wird ein Programmvorzustand in einen Programmnachzustand überführt. Die Regeln zur Auswertung von Termen eines Programms sind als induktiv definierte Menge formalisiert. Die Auswertung von Termen erfolgt „aggressiv“ [Ohe01]. Das bedeutet, dass die Auswertungsregeln so definiert sind, dass sie immer anwendbar sind. Durch die Anwendung „partieller“ Funktionen - zum Beispiel the_Addr bei der Auswertung des Call-Ausdrucks - können dadurch bei der Auswertung eines Terms undefinierte Situationen auftreten. Bei der Betrachtung einer Typzuverlässigkeitaussage über das modellierte Typsystem für ObjectTeams/Java in Abschnitt 9.1 können diese undefinierten Situationen jedoch durch Voraussetzungen der Typzuverlässigkeitsaussage ausgeschlossen werden, beispielsweise durch die Wohlgeformtheit eines Programms, oder sie werden durch das Exception-Konzept von Java behandelt.

Die Modellierung eines Exception-Konzepts für ObjectTeams/Java ist aus [KNO+02a] übernommen. Eine Exception ist Bestandteil eines erweiterten Programmzustands vom Typ *xstate* und stellt eine Referenz auf ein „vorallokiertes“ Exception-Objekt dar. Sobald eine Exception in einem Programmzustand enthalten ist, wird der Programmzustand bei der Ausführung eines Programms nicht weiter verändert, und die Exception wird bis zur Terminierung des Programms propagiert. Dieses Verhalten ist für alle Terme in einer einzigen Auswertungsregel formalisiert, so dass bei der Modellierung aller anderen Auswertungsregeln ein Exception-freier Programmvorzustand vorausgesetzt werden kann.

Der Typ *vals* fasst die Typen *val* und *val list* zusammen.

types *vals* = (*val* + *val list*)

Die Konstante arbitrary2 modelliert die Festlegung eines beliebigen Resultatwertes beim Auftritt einer Exception.

```
constdefs
```
arbitrary2 :: (('*s*,'*e*, '*t*)*sum*3 + '*el*) ⇒ *vals*
arbitrary2 ==
  sum_case (Inl o sum3_case (λx. Unit) (λx. arbitrary) (λx. arbitrary)) (λx. Inr arbitrary)

Die Auswertung eines Terms vom Typ *term* und die Produktion eines Resultatwertes vom Typ *vals*, bei der ein erweiterter Programmvorzustand vom Typ *xstate* in einen erweiterten Programmnachzustand

vom Typ *xstate* überführt wird, ist relativ zu einem Programm vom Typ *prog* als induktiv definierte Menge formalisiert. Folgende syntaktische Annotationen werden zur besseren Lesbarkeit definiert.

```
consts
  eval :: prog ⇒ (xstate ∗ term ∗ vals ∗ xstate)set

syntax
  eval :: [prog, xstate, term, vals ∗ xstate] ⇒ bool
                                        (_⊢_ −_−>> _ [51,82,60,82]81)
  exec :: [prog, xstate, stmt, xstate] ⇒ bool
                                        (_⊢_ −_−> _ [51,82,60,82]81)
  eval_ :: [prog, xstate, expr, val, xstate] ⇒ bool
                                        (_⊢_ −_−>_−> _ [51,82,60,82,82]81)
  evalt :: [prog, xstate, target, val, xstate] ⇒ bool
                                        (_⊢_ −_−>>_−> _ [51,82,60,82,82]81)
  evals :: [prog, xstate, expr list, val list, xstate] ⇒ bool
                                        (_⊢_ −_−#>_−> _ [51,82,60,51,82]81)

translations
  G ⊢ − t >> w_s' == (s, t, w_s') ∈ eval G
  G ⊢ − t >> (w, s') <= (s, t, (w,s')) ∈ eval G
  G ⊢ − t >> (w,x,s') <= (s, t, (w, x, s')) ∈ eval G
  G ⊢ − c -> s' == (s, Inl (In1 c), Inl Unit, s') ∈ eval G
  G ⊢ − c -> (x,s') <= (s, Inl (In1 c), Inl Unit, (x,s')) ∈ eval G
  G ⊢ − e -> v -> s' == (s, Inl (In2 e), Inl v, s') ∈ eval G
  G ⊢ − e -> v -> (x,s') <= (s, Inl (In2 e), Inl v, (x,s')) ∈ eval G
  G ⊢ − t >> v -> s' == (s, Inl (In3 t), Inl v, s') ∈ eval G
  G ⊢ − t >> v -> (x,s') <= (s, Inl (In3 t), Inl v, (x,s')) ∈ eval G
  G ⊢ − el −#> vl -> s' == (s, Inr el, Inr vl, s') ∈ eval G
  G ⊢ − el −#> vl -> (x,s') <= (s, Inr el, Inr vl, (x,s')) ∈ eval G
```

Die folgende Regel vereinheitlicht die Behandlung einer in einem erweiterten Programmvorzustand enthaltenen Exception und propagiert diese, ohne den Programmzustand weiter zu verändern.

$$\frac{}{\mathsf{G} \vdash (\mathsf{Some\ xc}, \mathsf{s}) \ - \ \mathsf{t} >> \ (\mathsf{arbitrary2\ t}, (\mathsf{Some\ xc}, \mathsf{s}))}$$

Die Regeln zur Ausführung von Anweisungen sind aus [KNO+02a] übernommen.

$$\frac{}{\mathsf{G} \vdash \mathsf{Norm\ s} \ - \mathsf{Skip} \to \mathsf{Norm\ s}}$$

$$\frac{\mathsf{G} \vdash \mathsf{Norm\ s0} \ - \mathsf{e} \to \mathsf{v} \to \mathsf{Norm\ s1}}{\mathsf{G} \vdash \mathsf{Norm\ s0} \ - \mathsf{Expr\ e} \to \mathsf{Norm\ s1}}$$

$$\frac{\mathsf{G} \vdash \mathsf{Norm\ s0} \ - \mathsf{c1} \to \mathsf{Norm\ s1} \qquad \mathsf{G} \vdash \mathsf{s1} \ - \mathsf{c2} \to \ \mathsf{s2}}{\mathsf{G} \vdash \mathsf{Norm\ s0} \ - \mathsf{c1;;\ c2} \to \ \mathsf{s2}}$$

$$\frac{\mathsf{G} \vdash \mathsf{Norm\ s0} \ - \mathsf{e} \to \ \mathsf{s1} \qquad \mathsf{G} \vdash \mathsf{s1} \ - (\mathsf{if\ (the\_Bool\ b)\ then\ c1\ else\ c2}) \to \ \mathsf{s2}}{\mathsf{G} \vdash \mathsf{Norm\ s0} \ - \mathsf{If(e)\ c1\ Else\ c2} \to \ \mathsf{s2}}$$

$$\frac{\mathsf{G} \vdash \mathsf{Norm\ s0} \ - \mathsf{e} \to \mathsf{b} \to \ \mathsf{s1} \qquad \neg(\mathsf{the\_Bool\ b})}{\mathsf{G} \vdash \mathsf{Norm\ s0} \ - \mathsf{While\ (e)c} \to \ \mathsf{s1}}$$

$$\frac{\mathsf{G} \vdash \mathsf{Norm\ s0} \ - \mathsf{e} \to \mathsf{b} \to \mathsf{s1} \qquad (\mathsf{the\_Bool\ b}) \qquad \mathsf{G} \vdash \mathsf{s1} \ - \mathsf{c} \to \mathsf{s2} \qquad \mathsf{G} \vdash \mathsf{s2} \ - \mathsf{While\ (e)c} \to \ \mathsf{s3}}{\mathsf{G} \vdash \mathsf{Norm\ s0} \ - \mathsf{While\ (e)c} \to \ \mathsf{s3}}$$

Die Regeln zur Auswertung einer Liste von Ausdrücken sind aus [KNO+02b] übernommen.

$$\frac{}{\mathsf{G} \vdash \mathsf{Norm\ s0} \ - [] \ -\#> [] \to \mathsf{Norm\ s0}}$$

$$\frac{G \vdash \text{Norm s0} \ -\ e \succ\ v \succ s1 \qquad G \vdash s1 \ -\ es \ -\#\succ\ vs \succ\ s2}{G \vdash \text{Norm s0} \ -\ (e\#es) \ -\#\succ\ (v\#vs) \ \succ s2}$$

Die Regeln zur Auswertung von Ausdrücken vom Typ *expr* und *target* werden im folgenden einzeln beschrieben.

Der Ausdruck NewC zur Instanziierung einer Klasse relativ zu einem Instanzkontext wird wie folgt ausgewertet.

- Durch Anwendung der Konstante new_Addr wird die Erzeugung einer neuen Referenz modelliert.
- Innerhalb der Definition des Prädikats newC_params ist die Bestimmung des Instanzkontextes, relativ zu dem ein Objekt erzeugt wird, die Bestimmung der Klasse, von der eine Instanz gebildet werden soll, und der Zugriff auf den Teamkontext, relativ zu dem die Attribute des neu zu erzeugenden Objekts bestimmt werden, formalisiert.
- Der Heap des Programmnachzustands wird mit dem neuen Objekt aktualisiert, wenn beim Erzeugen einer neuen Referenz keine Exception aufgetreten ist.

$$\frac{\begin{array}{c}h = \text{heap s} \qquad (a,x) = \text{new\_Addr h} \qquad \text{newC\_params G C s} = (\text{dic},C',\text{dynTC}) \\ h' = h(a \mapsto (|\text{tag} = (\text{dic}, C'), \text{values} = \text{init\_vals}(\text{var\_tys G } C' \text{ dynTC})|))\end{array}}{G \vdash \text{Norm s} \ -\ \text{NewC i C} \succ\ \text{Addr a} \succ\ \text{c\_hupd h}' \ (x,s)}$$

Die Konstante newC_params ist insofern „aggressiv“ definiert, als dass sie undefiniert ist, wenn die Variable TContext kein Teamobjekt referenziert, und keine die zu instanziierende Rollenklasse redefinierende, im dynamischen Teamkontext sichtbare Rollenklasse existiert. Bei der Betrachtung einer Typzuverlässigkeitsaussage über das Typsystem für ObjectTeams/Java in Abschnitt 9.2 treten diese beiden Fälle durch die Annahmen, dass ein Programmvorzustand bei der Betrachtung der Typzuverlässigkeit eines beliebigen Terms konform (siehe Abschnitt 9.1) zum statischen Environment ist, und das auszuwertende Programm wohlgeformt ist, nicht auf.

```
constdefs
  newC_params :: prog ⇒ tname ⇒ state ⇒ (dyn_ic * tname * tname)
  newC_params G C s ==
    if (is_bclass G C | is_tclass G C) then (DynTheVoid, C, C)
    else (DynTId (id_TContext s), the (rrc G (dyn_cls_TContext s) C), dyn_cls_TContext s)
```

Bei der Auswertung der Ausdrucks Cast und Inst müssen im Programmtext denotierte Rollentypen relativ zum dynamischen Teamkontext gebunden werden. Dies ist innerhalb der Modellierung der über die Typen *ref_ty* und *ty* mittels primitiver Rekursion definierten Funktionen calc_dyn_rty_of und calc_dyn_ty_of entsprechend der Definition der Konstanten newC_params „aggressiv“ formalisiert. Durch die Abbildung von im Programmtext denotierten statischen Typen auf dynamische instanzbasierte Typen wird die Semantik des Instanzkontextes der Typen definiert. Die Überprüfung, ob ein Rollenobjekt eine Instanz eines bestimmten Rollentyps darstellt, ist sowohl hinsichtlich der Klasse, als auch des Instanzkontextes, die den Rollentyp bilden, semantisch definiert.

```
consts
  calc_dyn_rty_of :: prog ⇒ state ⇒ ref_ty ⇒ dyn_ty
  calc_dyn_ty_of :: prog ⇒ state ⇒ ty ⇒ dyn_ty

primrec
  calc_dyn_rty_of G s NullT = DynNT
  calc_dyn_rty_of G s (ClassT i C) =
    (case i of
      TheVoid ⇒ DynClass DynTheVoid C
    | TThis ⇒ (if (is_bclass G C) then DynClass (DynTId (id_TContext s)) C
                else DynClass (DynTId (id_TContext s))
                       (the (rrc G (dyn_cls_TContext s) C))))

primrec
  calc_dyn_ty_of G s (PrimT pt) = DynPrimT pt
  calc_dyn_ty_of G s (RefT rt) = calc_dyn_rty_of G s rt
```

Innerhalb der Definition der Konstante fits ist die Überprüfung modelliert, ob der dynamische Typ eines Wertes zu dem im Programmtext denotierten und durch Anwendung der Konstanten calc_dyn_ty_of umtypisierten Referenztyp weitbar ist. Der Null-Typ ist zu jedem Referenztyp weitbar.

```
constdefs
  fits :: prog ⇒ state ⇒ val ⇒ ty ⇒ bool
  fits G s a' T ==
  (∃ rt. T = RefT rt) ⟶
    (a'=Null | (obj_ty (the ((heap s) (the_Addr a'))), calc_dyn_ty_of G s T) ∈ dyn_widen G)
```

Wenn der dynamische Typ des Wertes des Ausdrucks e nicht zum innerhalb des Programmtextes denotierten Typ T beziehungsweise zum relativ zum dynamischen Teamkontext umtypisierten Typ weitbar ist, was durch Anwendung der Konstanten fits überprüft wird, dann wird eine ClassCast-Exception geworfen. Es tritt also ein erwarteter Typfehler auf, da der Typ des Wertes des Ausdrucks e nicht die im Typ T beziehungsweise die im durch die Umtypisierung des Typs T relativ zum dynamischen Teamkontext bestimmten Typ enthaltenen Eigenschaften besitzt.

$$\frac{G \vdash Norm\ s0\ -e\Rightarrow\ v\Rightarrow\ (x1,s1) \qquad x2 = raise\_if\ (\neg(fits\ G\ s1\ v\ T))\ ClassCast\ x1}{G \vdash Norm\ s0\ -Cast\ T\ e\Rightarrow\ v\Rightarrow\ (x2,s1)}$$

Im Gegensatz zur Auswertung des Cast-Ausdrucks wird bei der Auswertung des Inst-Ausdrucks keine Exception geworfen, wenn der Wert des Ausdrucks e kein Element des Typs T beziehungsweise des durch die Umtypisierung des Typs T relativ zum dynamischen Teamkontext bestimmten Typ darstellt, sondern der boolesche Resultatwert wird dementsprechend bestimmt.

$$\frac{G \vdash Norm\ s0\ -e\Rightarrow\ v\Rightarrow\ s1 \qquad b = (v\ \neq\ Null\ \&\ (fits\ G\ (snd\ s1)\ v\ (RefT\ T)))}{G \vdash Norm\ s0\ -Inst\ e\ T\Rightarrow\ Bool\ b\Rightarrow\ s1}$$

Die Auswertung des Ausdrucks Lit besteht aus der Bestimmung des Wertes des Literals v.

$$\frac{}{G \vdash Norm\ s\ -Lit\ v\Rightarrow\ v\Rightarrow Norm\ s}$$

Der Zugriff auf eine Variable des dynamischen lokalen Environments wird durch Bestimmung des Wertes dieser Variablen ausgewertet.

$$\frac{}{G \vdash Norm\ s\ -LAcc\ v\Rightarrow\ (the\ ((locals\ s)\ v))\Rightarrow Norm\ s}$$

Der Ausdruck Tthis wird durch Zugriff auf den Wert der Variablen TContext des dynamischen lokalen Environments ausgewertet.

$$\frac{}{G \vdash Norm\ s\ -LAcc\ v\Rightarrow\ (the\ (val\_TContext\ s))\Rightarrow Norm\ s}$$

Der Variablen va des dynamischen lokalen Environments wird der Wert des Ausdrucks e zugewiesen, wenn bei dessen Auswertung keine Exception aufgetreten ist. Sonst wird diese propagiert und der Programmzustand nicht verändert.

$$\frac{G \vdash Norm\ s\ -e\Rightarrow\ v\Rightarrow\ (x,s') \qquad s'' = (if\ (x = None)\ then\ upd\_locals\ s'\ va\ v\ else\ s')}{G \vdash Norm\ s\ -va ::= e\Rightarrow\ v\Rightarrow (x,s'')}$$

Der Zugriff auf ein Attribut eines Objekts wird durch Bestimmung des innerhalb des Objektes auf dem Heap gespeicherten Wertes des Attributs ausgewertet. Handelt es sich beim dem Wert des Ausdrucks, über den auf das Attribut zugegriffen werden soll, um eine Null-Referenz, dann wird eine NullPointer-Exception geworfen. Wenn bei der Auswertung des Ausdrucks e eine Exception aufgetreten ist, wird diese propagiert.

$$\frac{G \vdash Norm\ s0\ -e\Rightarrow\ a'\Rightarrow\ (x1,s1) \qquad v = the\ ((values\ (the\ ((heap\ s1)\ (the\_Addr\ a'))))\ (fn,T)}{G \vdash Norm\ s0\ -\{T\}e..fn\Rightarrow\ v\Rightarrow\ (np\ a'\ x1,s1)}$$

Die Auswertung der Zuweisung eines Ausdrucks an ein Attribut eines Objekts findet wie folgt statt. Zuerst wird der Ausdruck, dessen Attribut ein Ausdruck zugewiesen werden soll, ausgewertet. Tritt dabei eine

NullPointer-Exception auf, wird diese propagiert. Ansonsten wird der zuzuweisende Ausdruck ausgewertet und an das Attribut fn des Ausdrucks e1 zugewiesen.

$$\frac{\begin{array}{c}G \vdash \text{Norm } s0 \; - \; e1 \Rightarrow a' \Rightarrow (x1,s1) \qquad a = \text{the\_Addr } a' \\ G \vdash (\text{np } a' \; x1,s1) \; - \; e2 \Rightarrow v \Rightarrow (x2,s2) \\ h = \text{heap } s2 \qquad \text{obj} = \text{the } (h \; a) \qquad h' = h \; (a \mapsto (\text{upd\_obj obj T fn } v))\end{array}}{G \vdash \text{Norm } s0 \; - \; \{T\}e1..fn := e2 \Rightarrow \; v \Rightarrow \; \text{c\_hupd } h' \; (x2, s2)}$$

Der Ausdruck zur Modellierung des Aufrufs einer Methode wird wie folgt ausgewertet.

- Der Ausdruck, auf dem die Methode aufgerufen werden soll, wird ausgewertet.
- Die Methodenparameterausdrücke werden ausgewertet.
- Die tatsächliche Klasse, von der aus die auszuführende Methode innerhalb der Funktion cmethod gesucht werden soll, wird durch die Konstante target bestimmt. Desweiteren wird innerhalb der Konstante target der Teamkontext bestimmt, relativ zu dem die Methode ausgeführt werden soll.
- Die durch Anwendung der Konstanten cmethod bestimmte Methodendefinition wird zur Aktualisierung des dynamischen lokalen Environments innerhalb der Konstanten init_lvars verwendet. Handelt es sich bei dem Wert des Ausdrucks, auf dem die Methode aufgerufen werden soll, um eine Null-Referenz, dann wird eine NullPointer-Exception geworfen. Ist bei der Auswertung der Methodenparameterausdrücke eine Exception aufgetreten, wird diese propagiert.
- Der Methodenresultatausdruck wird innerhalb des aktualisierten Programmzustands ausgewertet.
- Die Auswertung des Methodenaufrufausdrucks endet im Programmnachzustand, der sich aus der Exception- und der Heap-Komponente des letzten Programmzustands und aus dem Aufrufrahmen vor dem Aufruf der Methode zusammensetzt.

$$\frac{\begin{array}{c}G \vdash \text{Norm } s0 \; - \; e \gg\Rightarrow a' \Rightarrow s1 \qquad a = \text{the\_Addr } a' \\ G \vdash s1 \; - \; ps \; -\#\Rightarrow \; pvs \Rightarrow (x,(h,l)) \\ (C',\text{dynTC}) = \text{target mode } (h,l) \; a' \; G \; md \\ \text{cmethod } G \; C' \; \text{dynTC mn} = \text{Some } (\text{dynMd}, \text{Some dynPTs}, \text{Some dynM}) \\ G \vdash (\text{np } a' \; x, \; h, \; \text{init\_lvars } G \; \text{mode mn dynM } (h,l) \; a' \; pvs) \; - \; (\text{stmt } (\text{the } (\text{body dynM}))) \Rightarrow s3 \\ G \vdash s3 - \; (\text{res } (\text{the } (\text{body dynM}))) \Rightarrow \; v \Rightarrow \; (x4, s4)\end{array}}{G \vdash \text{Norm } s0 \; - \; \{md, mode\}e..mn(ps) \Rightarrow \; v \Rightarrow \; (x4, (\text{heap } s4, l))}$$

Innerhalb der Konstanten init_lvars werden abhängig vom Methodenaufrufmodus und der Art der Klasse, von der das *target* eine Instanz darstellt, die lokalen Variablen und Methodenparameter, This und der Teamkontext, relativ zu dem die Methode ausgeführt werden soll, gesetzt. Eine Aktualisierung des Teamkontextes findet beim Aufruf von Team- und Basismethoden statt.

```
constdefs
  init_lvars :: prog ⇒ inv_mode ⇒ mname ⇒ methd ⇒
                state ⇒ val ⇒ (val list) ⇒ dyn_inv_frame
  init_lvars G mode mn m s a' pvs ==
  dyn_inv_frame.make (obj_class (the ((heap s) (the_Addr a')))) mn mode
  (λ vn. case vn of
      VName v ⇒ (init_vals (table_of (lcl_vars (the (body m))))
                    ((params m) [↦]pvs)) vn
    | This ⇒ (if (¬(static m)) then Some a' else None)
    | TContext ⇒ (if (is_bclass G (obj_class (the ((heap s) (the_Addr a')))))
                 then None
                 else (if (is_tclass G (obj_class (the ((heap s) (the_Addr a')))))
                     then Some a' else val_TContext s)))

constdefs
  target :: inv_mode ⇒ state ⇒ val ⇒ prog ⇒ tname ⇒ (tname * tname)
  target m s a' G md ==
    (case m of
      SuperM ⇒ (if (is_bclass G md) then (md, md)
```

```
                else (md, dyn_cls_TContext s))
  | TSuperM ⇒ (md, dyn_cls_TContext s)
  | Static ⇒ (if (is_bclass G md) then (md, md)
                else (md, dyn_cls_TContext s))
  | IntVir ⇒ (if (is_bclass G (obj_class (the (heap s (the_Addr a'))))) |
                  is_tclass G (obj_class (the (heap s (the_Addr a')))))
               then (obj_class (the (heap s (the_Addr a'))),
                     obj_class (the (heap s (the_Addr a')))))
               else (obj_class (the (heap s (the_Addr a'))),
                     dyn_cls_TContext s)))
```

Bei der Auswertung der Ausdrücke Super und TSuper wird als Resultatwert der Wert verwendet, den die Variable This des dynamischen lokalen Environments enthält.

$$\frac{}{G \vdash Norm\ s\ -Super\gg\ (the\ (val\_this\ s))\ :\ Norm\ s}$$

$$\frac{}{G \vdash Norm\ s\ -TSuper\gg\ (the\ (val\_this\ s))\ :\ Norm\ s}$$

Die Auswertung des Ausdrucks TargetExpr besteht aus der Auswertung des Ausdrucks e.

$$\frac{G \vdash Norm\ s0\ -e\ :\ v\ :\ s1}{G \vdash Norm\ s0\ -TargetExpr\ e\gg\ v\ :\ s1}$$

Das Lemma eval_Inj_elim bringt zum Ausdruck, dass bei der Auswertung eines Terms der Wert Unit produziert wird, wenn es sich bei dem Term um eine Anweisung handelt. Wenn der Term einen Ausdruck oder eine Liste von Ausdrücken darstellt, dann wird ein Wert beziehungsweise eine Liste von Werten bei der Auswertung des Terms produziert. Das Lemma ist per *rule inversion* über die induktiv definierte Menge eval G nachgewiesen. Es wird bei der Analyse des *Confinement* von Rollenobjekten in Abschnitt 9.3 verwendet.

```
lemma eval_Inj_elim:
  G ⊢ s -t>> (w,s')
  ⟹ (case t of
        Inl se ⇒ (case se of
                    In1 s ⇒ w = Inl Unit
                  | In2 e ⇒ (∃ v. w = Inl v)
                  | In3 t ⇒ (∃ v. w = Inl v))
      | Inr es ⇒ (∃ v. w = Inr v))
```

Die folgenden Lemmata eval_no_xcpt und eval_xcpt werden bei der Betrachtung einer Typzuverlässigkeitsaussage über das modellierte Typsystem für ObjectTeams/Java benötigt. Sie sind per *rule inversion* über die induktiv definierte Menge eval G bewiesen und bringen folgendes zum Ausdruck.

- eval_no_xcpt: Ist nach der Auswertung eines Terms keine Exception im Programmnachzustand enthalten, dann war auch im Programmvorzustand keine enthalten.
- eval_xcpt: Ist bei der Auswertung eines Terms im Programmvorzustand eine Exception enthalten, dann ist diese auch im Programmnachzustand enthalten, und der Programmnachzustand ist gleich dem Programmvorzustand.

```
lemma eval_no_xcpt:
  G ⊢ (x, s) -t>> (v, x', s')
  ⟹ x' = None ⟶ x = None

lemma eval_xcpt:
  G ⊢ (Some xc, s) -t>> (v, x', s')
  ⟹ x' = Some xc & s' = s
```

9 Dynamische Analyse von ObjectTeams/Java

Es wird eine Typzuverlässigkeitsaussage über das in Kapitel 6 modellierte Typsystem für ObjectTeams/Java formuliert und untersucht mit dem Ziel, eine Aussage über die Typsicherheit von ObjectTeams/Java treffen zu können. Sie wird in Abschnitt 9.2 betrachtet. Es wird dargelegt, welche Teile dieser Aussage bewiesen sind, und wie noch offene Teile nachgewiesen werden könnten. Eine Bewertung dieser noch offenen Teile ist in Abschnitt 10.2 enthalten. Zur Formulierung der Typzuverlässigkeitsaussage wird in Abschnitt 9.1 erstens ein Prädikat über das Erhaltenbleiben von Objekten auf dem Heap bei der Ausführung von Programmen definiert. Zweitens werden Prädikate über die Konformität zwischen dynamischen Typen von Werten, die bei der Ausführung eines Programms durch Auswertung von Termen produziert werden, und den während der Wohlgeformtheitsprüfung dieses Programms ermittelten statischen Typen dieser Terme definiert.

Eine Aussage über das *Confinement* von Rollenobjekten ist in Abschnitt 9.3 formuliert und unter der Voraussetzung, dass die Typzuverlässigkeitsaussage über das Typsystem für ObjectTeams/Java vollständig bewiesen wird, nachgewiesen. Im Rahmen dieser Arbeit wird diese Typzuverlässigkeitsaussage weitgehend bewiesen.

9.1 Modellierung von Prädikaten zur Formulierung einer Typzuverlässigkeitsaussage

Das Prädikat heap_ex macht eine Aussage über das Erhaltenbleiben von Objekten auf dem Heap bei der Ausführung von Programmen. Sie beinhaltet eine Aussage über die Unveränderbarkeit der Klassen, von der Objekte Instanzen darstellen, und die Unveränderbarkeit der Kontexte, in denen sie enthalten sind. Diese Tatsache wird bei der Betrachtung der Typzuverlässigkeitsaussage über das Typsystem für ObjectTeams/Java benötigt, um zum Beispiel von der Existenz eines Objekts auf dem Heap in einem Programmvorzustand auf dessen Existenz in einem Programmnachzustand schließen zu können. Das Prädikat heap_ex und Aussagen darüber sind an [KNO+02a] und [KNO+02b] angelehnt.

```
constdefs
  heap_ex :: aheap ⇒ aheap ⇒ bool
  heap_ex h h' ==
    (∀ a obj. h a = Some obj −→ (∃ vs'. h' a = Some (obj (|values:=vs'|))))

lemma heap_exI:
  (∀ a obj. h a = Some obj −→ (∃ vs'. h' a = Some (obj (|values := vs'|))))
  ⟹ heap_ex h h'

lemma heap_ex_objD:
  [|heap_ex h h'; h a = Some obj|]
  ⟹ (∃ vs'. h' a = Some (obj (|values := vs'|)))

lemma heap_ex_refl [simp]: heap_ex h h

lemma heap_ex_NewC [simp]: h a = None ⟹ heap_ex h (h(a ↦ x))

lemma heap_ex_trans:
  [|heap_ex h h'; heap_ex h' h''|] ⟹ heap_ex h h''

lemma heap_ex_upd_obj:
  h a = Some obj ⟹ ∀ vs. heap_ex h (h(a ↦ (obj (|values := vs|))))
```

Zentral für die Formulierung einer Typzuverlässigkeitsaussage über das formalisierte Typsystem für ObjectTeams/Java ist die Definition einer Konformitätsbeziehung zwischen dynamischen Typen von Werten von Termen zur Laufzeit eines Programms und den bei der Wohlgeformtheitsüberprüfung dieses Programms ermittelten statischen Typen dieser Terme. Zur Bestimmung der tatsächlichen statischen Typen von Termen eines Programms wird der bei der Typisierung dieses Programms relativ zur Programmausführung bekannte dynamische Teamkontext verwendet. Dies setzt voraus, dass die Überprüfung der Wohlgeformtheit einer Klassendeklaration relativ zu dem Kontext, in dem sie enthalten ist, und die Wohlgeformtheit dieser Klassendeklaration in allen Teamkontexten entlang der Teamvererbungshierarchie, in denen ihre Eigenschaften wiederverwendet werden können, gilt (siehe Abschnitt 7.1.5).

Ein dynamischer Typ eines Wertes eines Terms ist in [KNO+02b] genau dann konform zum sta-

tischen Typ dieses Terms, wenn er zu diesem geweitet werden kann, also höchstens einen Subtyp des statischen Typs darstellt. Im Gegensatz zur Modellierung der Konformitätsprädikate in Java sind diese in ObjectTeams/Java über die Subtypbeziehung zwischen dynamischen Typen formuliert. Das hat zur Folge, dass die Prädikate neben der Aussage, ob der dynamische Typ eines Wertes eines Terms zur Laufzeit mindestens alle Eigenschaften des statischen Typs dieses Terms enthält, auch eine Aussage über das Enthaltensein von Objekten getroffen wird. Denn dynamische Typen stehen nur in einer Subtypbeziehung, wenn ihre Instanzkontexte, also die Kontexte, relativ zu denen sie gebunden werden, gleich sind. Das bedeutet auf Objektebene, dass die Objekte in derselben Instanz enthalten sind - gegebenenfalls in keiner.

Zur Definition der Konformität zwischen dynamischen Typen von Werten von Termen und den statischen Typen dieser Terme werden letztere mit Hilfe der Konstanten dyn_rty_of und dyn_ty_of auf ihre dynamischen Typen abgebildet. Dabei werden die Instanzkontexte, um die die statischen Typen angereichert sind, semantisch definiert. Ein Binden der die Typen bildenden Klassen abhängig vom dynamischen Teamkontext ist nicht nötig. Denn bei der Formulierung der Typzuverlässigkeitsaussage wird davon ausgegangen, dass der statische und der dynamische Teamkontext, relativ zu dem Typen gebunden werden, gleich sind. Dies wird innerhalb des Prädikats conf überprüft, das die Konformitätsprüfung zwischen einem Programmzustand und einem statischen Environment modelliert. (Es wird am Ende dieses Abschnittes definiert.)

```
consts
  dyn_rty_of :: prog ⇒ loc ⇒ ref_ty ⇒ dyn_ty
  dyn_ty_of :: prog ⇒ loc ⇒ ty ⇒ dyn_ty

primrec
  dyn_rty_of G id_TC (NullT) = DynNT
  dyn_rty_of G id_TC (ClassT i C) =
    (if (i=TThis) then DynClass (DynTId id_TC) C else DynClass DynTheVoid C)

primrec
  dyn_ty_of G id_TC (PrimT pt) = DynPrimT pt
  dyn_ty_of G id_TC (RefT rt) = dyn_rty_of G id_TC rt
```

Die folgenden Prädikate dienen zur Überprüfung der Konformität zwischen dynamischen Typen von Werten von Termen und statischen Typen dieser Terme auf der Ebene eines Wertes, einer Liste von Werten, eines Objekts, eines Heaps und eines Programmzustands.

Die Konformität zwischen dem dynamischen Typ eines Wertes eines Terms und dem statischen Typ dieses Terms wird durch das Prädikat vconf überprüft. Es wird gefordert, dass der dynamische Typ des Wertes durch Anwendung der Konstante typeof bestimmt werden kann. Wenn es sich bei dem statischen Typ um einen Rollentyp handelt, dann wird überprüft, ob der dynamische Typ des Wertes zu dem durch die Anwendung der Konstante dyn_ty_of bestimmten dynamischen Typ des statischen Typs geweitet werden kann. Dabei wird der Instanzkontext des statischen Typs auf die an das Prädikat vconf übergebene Identität des Teamkontextes abgebildet. Wenn es sich bei dem statischen Typ des Terms nicht um einen Rollentyp handelt, dann wird die Konformität zwischen dem dynamischen Typ des Wertes und dem durch die Anwendung der Konstante dyn_ty_of bestimmten dynamischen Typ des statischen Typs ohne Verwendung der übergebenen Identität des Teamkontextes überprüft. Denn diese wird erstens bei der Interpretation von Nicht-Rollentypen nicht verwendet, und zweitens ist der Teamkontext bei der Ausführung von Basismethoden nicht definiert.

```
consts
  vconf :: prog ⇒ aheap ⇒ loc ⇒ val ⇒ ty ⇒ bool
  res_conf :: prog ⇒ state ⇒ vals ⇒ tys ⇒ bool
  lconf :: prog ⇒ aheap ⇒ loc ⇒ (vname ⇒ val option) ⇒ (vname ⇒ ty option) ⇒ bool
  oconf :: prog ⇒ aheap ⇒ loc ⇒ obj ⇒ bool
  hconf :: prog ⇒ aheap ⇒ bool
  conf :: state ⇒ env ⇒ bool

defs
  vconf_def:
  vconf G h id_TC a' statT ==
    (∃ dynT. typeof (λa. option_map obj_ty (h a)) a' = Some dynT &
```

```
        (is_role_type G statT ⟶
          (dynT, dyn_ty_of G id_TC statT) ∈ dyn_widen G) &
        (¬(is_role_type G statT) ⟶
          (∀ bl. (dynT, dyn_ty_of G bl statT) ∈ dyn_widen G)))

types dyn_type = loc ⇒ dyn_ty option

consts
   typeof :: dyn_type ⇒ val ⇒ dyn_ty option

primrec
   typeof_Unit: typeof dt Unit = Some (DynPrimT Void)
   typeof_Bool: typeof dt (Bool b) = Some (DynPrimT Boolean)
   typeof_Intg: typeof dt (Intg i) = Some (DynPrimT Integer)
   typeof_Null: typeof dt Null = Some (DynRefT DynNullT)
   typeof_Addr: typeof dt (Addr a) = dt a
```

Das Prädikat res_conf modelliert die Konformitätsprüfung zwischen einem dynamischen Typ eines Wertes eines Terms und dem statischen Typ dieses Terms und zwischen einer Liste dynamischer Typen einer Liste dynamischer Werte von Termen und einer Liste statischer Typen dieser Terme. Das Prädikat wird bei der Überprüfung der Konformität zwischen dynamischen Typen von bei der Auswertung von Termen produzierten Werten und den statischen Typen dieser Terme verwendet. Zur Abbildung von statischen auf dynamische Typen wird eine Identität des Teamkontextes benötigt. Dafür wird die innerhalb des dynamischen lokalen Environments in der Variablen TContext enthaltene Referenz auf das akutelle Teamobjekt verwendet. Auf diese wird mit Hilfe der Konstanten id_TContext zugegriffen. Wenn der ausgewertete Term aus einer Basismethode stammt, dann ist im dynamischen lokalen Environment keine Referenz auf ein Teamobjekt enthalten, relativ zu dessen Identität Rollentypen interpretiert werden könnten. Innerhalb des Lemmas occurence_of_role_types (siehe Abschnitt 7.2.1) ist jedoch nachgewiesen, dass in Basisklassenmethoden keine Rollentypen vorkommen. Der Zugriff auf den Wert der Variablen TContext ist innerhalb von Basisklassenmethoden zwar undefiniert, aber das Resultat des Zugriffs wird innerhalb des Prädikats vconf nicht verwendet.

```
   res_conf_def:
   res_conf G s v T ==
     (case T of
       Inl t ⇒ vconf G (heap s) (id_TContext s) (the_Inl v) t
       | Inr ts ⇒ list_all2 (vconf G (heap s)) (id_TContext s)) (the_Inr v) ts)
```

Das Prädikat lconf modelliert die Konformitätsprüfung zwischen einer Liste von dynamischen Typen von Werten von Termen und einer Liste statischer Typen dieser Terme. Es wird für die Überprüfung der Konformität zwischen dynamischen Typen von im dynamischen lokalen Environment enthaltenen Werten von Variablen und den im statischen lokalen Environment enthaltenen statischen Typen dieser Variablen verwendet. Die Konformität zwischen dem dynamischen Typ des Wertes von this und dem statischen Typ von this wird separat innerhalb des Prädikats conf überprüft. Denn der statische Typ von this wird im Gegensatz zu allen anderen im statischen lokalen Environment enthaltenen Typen nicht relativ zum aktuellen Teamkontext gebunden (siehe Abschnitt 7.1.3).

```
   lconf_def:
   lconf G h id_TC vs Ts ==
     (∀ n T. Ts n = Some T & n  = This ⟶
       (∃ v. vs n = Some v & vconf G h id_TC v T))
```

Das Prädikat oconf modelliert die Überprüfung der Konformität zwischen den dynamischen Typen von Attributwerten eines Objekts und den deklarierten Attributtypen. Die Attributtypen eines Objekts werden mit Hilfe der Konstante var_tys (siehe Abschnitt 8.1.1) bestimmt. Wenn es sich bei dem Objekt um ein Rollenobjekt handelt, dann wird zum Binden der im Programmtext deklarierten Attributtypen die die Rollenklasse umschließende Teamklasse verwendet, von der das Rollenobjekt eine Instanz darstellt. Ansonsten wird die Klasse verwendet, von der das Objekt eine Instanz darstellt. Desweitern wird innerhalb des Prädikats oconf überprüft, ob das Objekt einen korrekten dynamischen Typ besitzt. Dazu wird das Prädikat is_obj_ty verwendet (siehe Abschnitt 8.1.1).

```
oconf_def:
oconf G h id_TC obj ==
  (∀ n T. (var_tys G (obj_class obj)
    (if (is_rclass G (obj_class obj))
    then encl_class (the (rclass G (obj_class obj)))
    else obj_class obj)) n = Some T −→
      (∃ v. (values obj) n = Some v & vconf G h id_TC v T)) &
  is_obj_ty G obj
```

Das Prädikat hconf modelliert die Überprüfung der Konformität zwischen einem Heap h und dem Programm G, das ausgeführt wird. Es wird überprüft, ob alle auf dem Heap vorhandenen Objekte konform sind. Dabei wird bei Rollenobjekten die Referenz auf das Teamobjekt, in dem sie enthalten sind, bei Teamobjekten die Referenz auf sie selbst und bei Basisobjekten eine beliebige Identität des Teamkontextes zur Interpretation von statischen Typen verwendet.

```
hconf_def:
hconf G h ==
  (∀ a obj. h a = Some obj −→
    ((is_rclass G (obj_class obj) & oconf G h (the (tid (obj_dic obj))) obj) |
    (is_tclass G (obj_class obj) & oconf G h a obj) |
    (is_bclass G (obj_class obj) & (∀ bl. oconf G h bl obj))))
```

Das Prädikat conf modelliert die Überprüfung der Konformität zwischen einem Programmzustand und einem statischen Environment. Dabei wird folgendes überprüft.

1. Die Konformität zwischen dem Heap des Programmzustands und dem statischen Environment wird überprüft.

2. Die Konformität zwischen dem dynamischen lokalen Environment und dem statischen lokalen Environment wird abhängig von der Art der Klasse überprüft, in der die ausgeführte Methode definiert ist.

3. Stellt die aktuelle Klasse eine Basisklasse dar, dann wird die Konformität zwischen dem dynamischen lokalen Environment und dem statischen lokalen Environment für alle beliebigen Identitäten von Teamkontexten überprüft. Denn der Teamkontext ist bei der Ausführung von Basisklassenmethoden nicht definiert. Wenn eine Instanzmethode ausgeführt wird, wird zusätzlich die Konformität zwischen dem dynamischen Typ des Wertes von this und dem statischen Typ von this überprüft.

4. Wenn die aktuelle Klasse eine Rollenklasse darstellt, dann wird die Konformität zwischen dem dynamischen und dem statischen lokalen Environment überprüft. Dabei wird die in der Variablen TContext enthaltene Referenz auf das aktuelle Teamobjekt verwendet. Die Konformität zwischen dem dynamischen Typ des Wertes von this und seinem statischen Typ wird durch die explizite Bindung des statischen Typs überprüft. Denn der der Variablen This zugeordnete Typ wird im statischen lokalen Environment nicht abhängig vom aktuellen Teamkontext gebunden.

5. Handelt es sich bei der aktuellen Klasse um eine Teamklasse, dann wird die Konformität zwischen dem dynamischen und dem statischen lokalen Environment und dem dynamischen und dem statischen Typ von this überprüft. Dabei wird die in der Variablen TContext enthaltene Referenz auf das aktuelle Teamobjekt verwendet.

6. Wenn im statischen lokalen Environment ein Teamkontext definiert ist, dann wird zusätzlich zur Konformitätsüberprüfung zwischen dem dynamischen Typ des in der Variablen TContext enthaltenen Wertes und dem statischen Typ, der dieser Variablen zugewiesen ist, überprüft, ob der Wert eine Referenz auf ein Teamobjekt auf dem Heap darstellt. (Denn der Null-Typ ist zu allen Referenztypen weitbar.) Es wird überprüft, ob dieses Teamobjekt eine Instanz der Teamklasse ist, die den statischen Typ des der Variablen TContext zugewiesenen Typs bildet.

   Diese Überprüfung ist notwendig, damit die Bindungen von im Programmtext denotierten Typen hin zu dynamischen Typen und die Interpretation von statischen Typen hin zu dynamischen Typen definiert ist.

Es ist durch Nachweis des Lemmas tcontext_eq mittels *rule induction* über die Menge eval G gezeigt, dass der Teamkontext bei der Auswertung eines beliebigen Terms ausgehend von einem Programmvorzustand gleich dem Teamkontext im Programmnachzustand ist.

```
conf_def:
  conf s E ==
    (hconf (prg E) (heap s) &
    (is_bclass (prg E) (act_cls E) | is_tclass (prg E) (act_cls E) | is_rclass (prg E) (act_cls E)) &
    (is_bclass (prg E) (act_cls E) −→
      (∀ bl. lconf (prg E) (heap s) bl (locals s) (lcl E)) &
      ((lcl E) This ≠ None −→
        (∀ bl. vconf (prg E) (heap s) bl (the ((locals s) This)) (the ((lcl E) This))))) &
    (is_rclass (prg E) (act_cls E) −→
      lconf (prg E) (heap s) (id_TContext s) (locals s) (lcl E) &
      (∃ C. (lcl E) This = Some (Class TThis C) & is_rclass (prg E) C) &
        vconf (prg E) (heap s) (id_TContext s) (the ((locals s) This))
          (Class (the (ic (the ((lcl E) This)))) (the (rrc (prg E)
            (the (cls (the (tcontext E)))) (the (cls (the ((lcl E) This)))))))) &
    (is_tclass (prg E) (act_cls E) −→
      lconf (prg E) (heap s) (id_TContext s) (locals s) (lcl E) &
      vconf (prg E) (heap s) (id_TContext s) (the ((locals s) This)) (the ((lcl E) This))) &
    (tcontext E ≠ None −→
      (locals s) TContext ≠ None &
      (∃ a. (locals s) TContext = Some (Addr a) & (heap s) a ≠ None) &
      the (cls (the (tcontext E))) = dyn_cls_TContext s))

lemma tcontext_eq [rule_format]:
  G ⊢ (x,h,l) −t >> (v,x',h',l')
  ⟹ wf_prog G −→ (∀ L T. (G,L) |= t :: T −→
       ((dyn_lenv l) TContext) = ((dyn_lenv l') TContext))
```

9.2 Typzuverlässigkeitsaussage über das modellierte Typsystem für ObjectTeams/Java

Es wird eine Typzuverlässigkeitsaussage über das in Kapitel 6 modellierte Typsystem für ObjectTeams/Java formuliert. Sie unterscheidet sich aufgrund der Definition der Konformitätsprädikate über die Subtyprelation zwischen dynamischen instanzbasierten Typen von der in [KNO+02b] enthaltenen Typzuverlässigkeitsaussage über das formalisierte Typsystem für Java (siehe Abschnitt 9.1).

Die Typzuverlässigkeitsaussage eval_type_sound_ot über das modellierte Typsystem für ObjectTeams/Java bringt folgendes zum Ausdruck. Wenn ein beliebiger Term t relativ zu einem Programm G ausgehend von einem Programmvorzustand (x,(h,l)) ausgewertet wird, dabei ein Resultatwert v produziert und der Programmvorzustand (x,(h,l)) in einen Programmnachzustand (x',(h',l')) überführt wird und das Programm G wohlgeformt ist, dann soll für alle statischen lokalen Environments L, für die der Programmvorzustand ohne Excpetion-Komponente (h,l) konform zum statischen Environment (G,L) ist, und für alle Typen T, für die der Term t wohlgetypt ist, gelten, dass alle Objekte auf dem Heap h des Programmvorzustands höchstens mit veränderten Attributwerten auf dem Heap des Programmnachzustands h' vorhanden sind, dass der Programmnachzustand ohne Excpetion-Komponente (h',l') konform zum statischen Environment (G,L) ist und, falls bei der Auswertung des Terms t keine Exception aufgetreten ist, dass der dynamische Typ des bei der Auswertung des Terms t produzierten Wertes v konform zum statischen Typ T des Terms t ist.

```
theorem eval_type_sound_ot [rule_format]:
  G ⊢ (x,(h,l)) − t >> (v,x',(h',l')) ⟹
  (wf_prog G −→ (∀ L . (conf (h,l) (G,L)) −→ (∀ T. (G,L) |= t :: T −→
    (heap_ex h h' &
    (conf (h',l') (G,L)) &
    (x'=None −→ res_conf G (h',l') v T)))))
```

Im folgenden werden die bewiesenen und die noch offenen Teile der Typzuverlässigkeitsaussage

eval_type_sound_ot skizziert. Sie ist mittels *rule induction* über die Menge eval G untersucht. Die Beweissubziele werden analog zu den durch die Anwendung der Induktionsregel der Menge eval G generierten Beweissubzielen aufgelistet.

1. Exception. Die Typzuverlässigkeit des Typsystems für ObjectTeams/Java beim Auftritt einer Exception ist mit Hilfe der Induktionsvoraussetzungen nachgewiesen. Denn der Programmzustand wird bei Auftritt einer Exception nicht verändert, ein unveränderter Heap erfüllt das Prädikat heap_ex (siehe Lemma heap_ex_refl in Abschnitt 9.1), und bei Auftritt einer Exception (x'≠None) wird die Konformität zwischen dem dynamischen Typ des bei der Auswertung des Terms produzierten Wertes und dem statischen Typ dieses Terms nicht betrachtet.

2. Skip. Die Typzuverlässigkeit des Typsystems für ObjectTeams/Java bei der Ausführung der Anweisung Skip ist nachgewiesen. Ein unveränderter Heap erfüllt das Prädikat heap_ex. Die Konformität zwischen dem Programmnachzustand, der gleich dem Programmvorzustand ist, und dem statischen Environment gilt per Induktionsvoraussetzung. Die Konformität zwischen dem bei der Ausführung der Anweisung Skip produzierten Wert Unit und dem statischen Typ der Anweisung PrimT Void ist durch den Nachweis des Lemmas res_conf_Unit_Void gezeigt.

3. Expr. Die Typzuverlässigkeit des Typsystems für ObjectTeams/Java beim Einpacken eines Ausdrucks in eine Anweisung ist per Induktionsvoraussetzung und durch den Nachweis des Lemmas res_conf_Unit_Void gezeigt.

4. Comp. Die Typzuverlässigkeit des Typsystems für ObjectTeams/Java bei der Komposition von Anweisungen ist per Induktionsvoraussetzung und durch den Nachweis des Lemmas heap_ex_trans gezeigt.

5. Cond. Die Typzuverlässigkeit des Typsystems für ObjectTeams/Java bei der Ausführung einer konditionalen Anweisung ist per Induktionsvoraussetzung und durch den Nachweis des Lemmas heap_ex_trans gezeigt.

6. LoopF. Die Typzuverlässigkeit des Typsystems für ObjectTeams/Java bei der Auswertung einer Schleifenbedingung ist per Induktionsvoraussetzung und durch den Nachweis des Lemmas res_conf_Unit_Void gezeigt.

7. LoopT. Die Typzuverlässigkeit des Typsystems für ObjectTeams/Java bei der Ausführung einer Schleife ist per Induktionsvoraussetzung und durch den Nachweis des Lemmas heap_ex_trans gezeigt.

8. NewC. Die Typzuverlässigkeit des Typsystems für ObjectTeams/Java bei der Instanziierung einer Klasse ist nicht vollständig nachgewiesen, sondern mit Hilfe von fünf Axiomen in nachzuweisende Aussagen aufgespalten. Diese Aussagen erfordern den Nachweis der Objektkonformität eines mit Default-Werten initialisierten neu erzeugten Objekts, den Nachweis der Konformität zwischen dem um das neu erzeugte Objekt aktualisierten Programmzustand und dem statischen Environment, und den Nachweis der Konformität zwischen dem dynamischen Typ der Referenz auf das neu erzeugte Objekt und dem statischen Typ des Ausdrucks NewC. Dabei müssen die während der statischen Analyse durch die Konstante check_type und die bei der Ausführung des Programms durch die Konstante newC_params bestimmten Klassen verglichen werden.

9. Cast. Die Typzuverlässigkeit des Typsystems für ObjectTeams/Java bei der Auswertung des Cast-Ausdrucks ist durch den Nachweis des Lemmas eval_ts_Cast und per Induktionsvoraussetzung gezeigt.

10. Inst. Die Typzuverlässigkeit des Typsystems für ObjectTeams/Java bei der Auswertung des Ausdrucks Inst ist per Indunktionsvoraussetzung und durch den Nachweis des Lemmas res_conf_Bool_Boolean gezeigt. (Dieses bringt zum Ausdruck, dass der Typ eines booleschen Wertes konform zum Typ PrimT Boolean ist.)

11. Lit. Die Typzuverlässigkeit des Typsystems für ObjectTeams/Java bei der Auswertung des Zugriffs auf ein Literal ist durch den Nachweis des Lemmas heap_ex_refl, per Induktionsvoraussetzung und durch den Nachweis des Lemmas eval_ts_Lit gezeigt.

12. LAcc. Die Typzuverlässigkeit des Typsystems für ObjectTeams/Java beim Zugriff auf eine Variable des lokalen Environments ist durch den Nachweis des Lemmas heap_ex_refl, per Induktionsvoraussetzung und durch den Nachweis des Lemmas eval_ts_LAcc gezeigt. Letzteres ist durch den Nachweis der Aussagen res_conf_check_type und res_conf_check_local_type gezeigt, die per Fallunterscheidung über den zu überprüfenden Typ des statischen lokalen Environments bewiesen sind.

    Das Lemma check_local_type_eq verdeutlicht, dass die Konstante check_local_type auch als Prädikat hätte definiert werden können.

13. Tthis. Die Typzuverlässigkeit des Typsystems für ObjectTeams/Java bei der Auswertung des Ausdrucks Tthis ist durch die Nachweise der Lemmata heap_ex_refl und eval_ts_Tthis und per Induktionsvoraussetzung gezeigt.

14. LAss. Die Typzuverlässigkeit des Typsystems für ObjectTeams/Java bei der Zuweisung eines Wertes an ein Variable des lokalen Environments ist durch den Nachweis des Lemmas eval_ts_LAss gezeigt. Dieses ist durch die Nachweise der Lemmata check_local_type_LAss, conf_upd_locals und res_conf_locals_lcl nachgewiesen.

15. FAcc. Die Typzuverlässigkeit des Typsystems für ObjectTeams/Java bei einem Attributzugriff ist offen und setzt voraus, dass die innerhalb des Wohlgeformtheitsprädikats für eine Klassendeklaration geforderten Einschränkungen bezüglich Attributdeklarationen für alle möglichen Teamkontexte entlang der entsprechenden Teamklassenvererbungshierarchie, in denen diese geerbt werden können, nachgewiesen ist. Eine weitere Schwierigkeit stellt die Behandlung des zweidimensionalen Erbens von Rollenklassen entlang der `extends`- und der impliziten Vererbungshierarchie dar.

16. FAss. Die Typzuverlässigkeit des Typsystems für ObjectTeams/Java bei der Zuweisung eines Wertes an ein Attribut eines Objekts ist nicht nachgewiesen, sondern mit Hilfe eines Axioms formuliert. Dieses enthält eine Aussage über die Konformität des aktualisierten Programmzustands unter der Voraussetzung, dass der nicht aktualisierte Programmzustand konform ist, und der dem Attribut zugewiesene Wert konform zu einem Typ ist, der zum Typ des Attributs geweitet werden kann. Entsprechend der Betrachtung der Typzuverlässigkeit des Typsystems für ObjectTeams/Java bei einem Attributzugriff wird eine Aussage über die Wohlgeformtheit von Attributdeklarationen entlang der entsprechenden Teamklassenvererbungshierarchie benötigt.

17. Call. Die Typzuverlässigkeit des Typsystems für ObjectTeams/Java beim Aufruf einer Methode ist offen und setzt den Nachweis der Wohlgeformtheit von Methodendeklarationen in allen Kontexten, in denen diese geerbt werden können, voraus. Vier Axiome sind zur Zerlegung des zu führenden Beweises formuliert. Erstens muss eine Aussage bezüglich der Wohlgetyptheit des Methodenrückgabeausdrucks und Einschränkungen beim Überschreiben von Methoden gezeigt werden. Zweitens muss die Weitbarkeit des dynamischen Typs des Resultatwertes des Methodenaufrufs zum statischen Typ des Methodenaufrufausdrucks gezeigt werden. Dies sollte mittels der Transitivität der Subtyprelation für dynamische und statische Typen und Fallunterscheidungen und Einschränkungen bezüglich des Vorkommens von Rollentypen als Typen von Methodenparametern gezeigt werden können. Drittens muss die Konformität des aktualisierten Programmzustands zum aktualisierten statischen Environment nachgewiesen werden. Viertens muss die Konformität zwischen dem Programmnachzustand und dem statischen Environment gezeigt werden.

18. Super. Die Typzuverlässigkeit des Typsystems für ObjectTeams/Java bei der Auswertung des Ausdrucks Super ist durch die Nachweise der Lemmata heap_ex_refl und res_conf_check_type und per Induktionsvoraussetzung gezeigt.

19. TSuper. Die Typzuverlässigkeit des Typsystems für ObjectTeams/Java bei der Auswertung des Ausdrucks TSuper ist durch die Nachweise der Lemmata heap_ex_refl und res_conf_check_type und per Induktionsvoraussetzung gezeigt.

20. TargetExpr. Die Typzuverlässigkeit des Typsystems für ObjectTeams/Java beim Einpacken eines Ausdrucks in einen Term vom Typ *target* ist per Induktionsvoraussetzung gezeigt.

21. Nil. Die Typzuverlässigkeit des Typsystems für ObjectTeams/Java bei der Auswertung einer leeren Liste von Ausdrücken ist durch den Nachweis des Lemmas heap_ex_refl und per Indunktionsvoraussetzung gezeigt.

22. Cons. Die Typzuverlässigkeit des Typsystems für ObjectTeams/Java bei der Auswertung einer nicht leeren Liste von Ausdrücken ist durch den Nachweis des Lemmas res_conf_Cons und per Induktionsvoraussetzung gezeigt.

Im folgenden werden einige bei der Betrachtung der Typzuverlässigkeitsaussage für ObjectTeams/Java nachgewiesene Aussagen vorgestellt. Eine Bewertung der Aussagen, die noch nicht nachgewiesen sind, ist in Abschnitt 10.2 enthalten.

9.2.1 Nachgewiesene Lemmata bei der Untersuchung der Typzuverlässigkeitsaussage über das Typsystem für ObjectTeams/Java

Cast-Ausdruck Das Lemma eval_ts_Cast bringt zum Ausdruck, dass der dynamische Typ des bei der Auswertung des Cast-Ausdrucks produzierten Wertes zum innerhalb des Programmtexts denotierten und relativ zum aktuellen Teamkontext gebundenen Typ konform ist. Das Lemma eval_ts_Cast ist mittels Fallunterscheidungen über die cast G-Typrelation und den Wert v, durch die Anwendung von Hilfsaussagen über das Prädikat res_conf und durch die Anwendung des Lemmas res_conf_Addr nachgewiesen.

```
lemma eval_ts_Cast:
  [|((None, a, b), Inl (In2 e), Inl v, x1, aa, ba) |= eval G; wf_prog G;
  (G, L) |= Inl (In2 e)::Inl S; check_type (G, L) T = Some T'; (S, T') |= cast (fst (G, L));
  ¬¬fits G (aa, ba) v T; x1 = None; conf (aa, ba) (G, L); res_conf G (aa, ba) (Inl v) (Inl S)|]
  ⟹ res_conf G (aa, ba) (Inl v) (Inl T')
```

Das Lemma res_conf_Addr_Class bringt zum Ausdruck, dass der dynamische Typ des durch die Referenz l referenzierten Objekts konform zum durch die Konstante check_rtype bestimmten Typ ist, wenn der dynamsiche Typ des durch die Referenz referenzierten Objekts weitbar zum durch die Konstante calc_dyn_rty_of bestimmten Typ ist. Dabei muss erstens unterschieden werden, ob der im Programmtext denotierte Typ tatsächlich abhängig vom Teamkontext gebunden wird. Zweitens muss, wenn der Typ durch eine Rollenklasse gebildet ist, unterschieden werden, ob diese bereits eine aus dem aktuellen Teamkontext ist oder nicht. Dies ist durch den Nachweis des Lemmas rrc_eq erfasst.

```
lemma res_conf_Addr_Class:
  [|wf_prog G; (G, L) |= Inl (In2 e)::Inl (Class i C'); res_conf G (aa, ba) (Inl (Addr l)) (Inl (Class i C'));
  check_rtype (G, L) t = Some (Class i C); conf (aa, ba) (G, L);
  (obj_ty (the (heap (aa, ba) (the_Addr (Addr l)))), calc_dyn_rty_of G (aa, ba) t) ∈ dyn_widen G|]
  ⟹ res_conf G (aa, ba) (Inl (Addr l)) (Inl (Class i C))
```

Ausdruck zum Zugriff auf ein Literal Das Lemma eva_ts_Lit bringt zum Ausdruck, dass der Typ eines primitiven Wertes konform zu seinem statischen primitiven Typ ist. Es ist mittels Fallunterscheidungen über den Wert v nachgewiesen.

```
lemma eval_ts_Lit:
  typeof (λv. None) v = Some (DynPrimT pt)
  ⟹ res_conf G (fst (snd (None, aa, ba)), snd (snd (None, aa, ba))) (Inl v) (Inl (PrimT pt))
```

Ausdruck zum Zugriff auf eine Variable des lokalen Environments Die Konformität zwischen dem dynamischen Typ des Wertes einer Variablen v aus dem dynamischen lokalen Environment und dem mit Hilfe der Konstanten check_type und check_local_type relativ zum aktuellen Teamkontext gebundenen statischen Typ der Variablen v ist durch das Lemma eval_ts_LAcc formuliert und durch die Anwendung des Lemmas res_conf_check_type, wenn die Variable v This darstellt, und ansonsten durch die Anwendung des Lemmas res_conf_check_local_type gezeigt. Das Lemma res_conf_check_type ist durch Anwendung verschiedener Hilfslemmata mit Fallunterscheidungen über den zu bindenden statischen Typ der Variablen des lokalen Environments gezeigt. Das Lemma res_conf_check_local_type ist unter Verwendung der Konformität des Programmzustandes (None,a,b) nachgewiesen.

```
lemma eval_ts_LAcc:
  [|wf_prog G; conf (fst (snd (None, aa, ba)), snd (snd (None, aa, ba))) (G, L);
  (lcl (G, L)) v = Some Ta; v ≠ TContext; T = Inl T';
  v = This ⟶ check_type (G, L) Ta = Some T' &
       T' = Class i C & ¬ static_method (G, L);
```

```
    v ≠ This ⟶ check_local_type (G, L) Ta = Some T' |]
    ⟹ res_conf G (fst (snd (None, aa, ba)), snd (snd (None, aa, ba)))
          (Inl (the (locals (aa, ba) v))) (Inl T')

lemma res_conf_check_type:
    [|wf_prog G; conf (fst (snd (None, aa, ba)), snd (snd (None, aa, ba))) (G, L);
    (lcl (G, L)) This = Some Ta; check_type (G, L) Ta = Some (Class i C)|]
    ⟹ res_conf G (aa, ba) (Inl (the (dyn_lenv ba This))) (Inl (Class i C))

lemma res_conf_check_local_type:
    [|wf_prog G; conf (fst (snd (None, a, b)), snd (snd (None, a, b))) (G, L);
    (lcl (G, L)) v = Some Ta; v ≠ TContext; T = Inl T'; fst (None, a, b) = None;
    v ≠ This; check_local_type (G, L) Ta = Some T' |]
    ⟹ res_conf G (a, b) (Inl (the (dyn_lenv b v))) (Inl T')

lemma check_local_type_eq:
    check_local_type (G,L) T = Some T' ⟹ T = T'
```

Ausdruck Tthis Die Konformität zwischen dem dynamischen Typ des Wertes von Tthis, der die Referenz auf das aktuelle Teamobjekt darstellt, und dem statischen Typ von Tthis ist innerhalb des Lemmas eval_ts_Tthis formuliert. Dieses ist durch Anwendung des Lemmas vconf_TContext nachgewiesen. Durch Nachweis des Lemmas vconf_TContext ist eben diese Konformität bewiesen.

```
lemma eval_ts_Tthis:
    [|wf_prog G; conf (fst (snd (None, a, b)), snd (snd (None, a, b))) (G, L);
    type_Tthis (G, L) = Some T'; T = Inl T'; fst (None, a, b) = None|]
    ⟹ res_conf G (fst (snd (None, a, b)), snd (snd (None, a, b))) (Inl (the (val_TContext (a, b)))) T

lemma vconf_TContext:
    [|is_rclass G (act_class L); check_tcontext (G, L) = Some yb;
    wf_prog G; lenv L This = Some y; is_class_type G y; cls y = Some (act_class L);
    hconf (fst (G, L)) (heap (a, b)); ic_correct G (the (ic y)) (act_class L);
    lconf (fst (G, L)) (heap (a, b)) (id_TContext (a, b)) (locals (a, b)) (lcl (G, L));
    (lcl (G, L)) TContext ≠ None ⟶
        locals (a, b) TContext ≠ None &
        (∃ aa. locals (a, b) TContext = Some (Addr aa) & heap (a, b) aa ≠ None) &
    the (cls (the ((lcl (G, L)) TContext))) = dyn_cls_TContext (a, b)|]
    ⟹ vconf G a (the_Addr (the (dyn_lenv b TContext)))
          (the (dyn_lenv b TContext))
             (Class TheVoid (encl_class (the (rclass G (act_class L)))))
```

Zuweisung eines Ausdrucks an eine Variable des lokalen Environments Die Typzuverlässigkeit des Typsystems für ObjectTeams/Java beim Zugriff auf eine Variable des lokalen Environments ist durch den Nachweis des Lemmas eval_ts_LAss nachgewiesen. Es bringt zum einen zum Ausdruck, dass der aktualisierte Programmzustand konform zum statischen Environment ist. Dies ist durch den Nachweis des Lemma conf_upd_locals gezeigt. Zum anderen ist gezeigt, dass der dynamische Typ des Wertes des an die Variable zugewiesenen Ausdrucks konform zum statischen Typ des Zuweisungsausdrucks ist. Dies ist durch den Nachweis des Lemma res_conf_locals_lcl gezeigt.

Das Lemma conf_upd_locals ist durch den Nachweis einer Anzahl von Hilfslemmata für die unterschiedlichen Klassenarten und durch die Formulierung von Aussagen über den Zusammenhang von innerhalb einer Subtypbeziehung stehenden statischen und dynamischen Typen nachgewiesen. Das Lemma res_conf_locals_lcl ist durch Auffalten von Definitionen nachgewiesen.

```
lemma eval_ts_LAss:
    [|wf_prog G; (G, L) |= Inl (In2 e)::Inl S; (S, T') ∈ widen G;
    va ≠ This; va ≠ TContext; lenv L va = Some Tb; check_local_type (G, L) Tb = Some T';
    conf (aa, ba) (G, L); res_conf G (aa, ba) (Inl v) (Inl S)|]
    ⟹ conf (aa, ba(| dyn_lenv := locals (aa, ba)(va ↦ v) |)) (G, L) &
        res_conf G (aa, ba(| dyn_lenv := locals (aa, ba)(va ↦ v) |)) (Inl v) (Inl S)
```

```
lemma conf_upd_locals:
  [|wf_prog G; (G, L) |= Inl (In2 e)::Inl S; T' ≠ NT; (S, T') ∈ widen G;
  va ≠ This; va ≠ TContext; lenv L va = Some Tb; check_local_type (G, L) Tb = Some T';
  conf (aa, ba) (G, L); res_conf G (aa, ba) (Inl v) (Inl S)|]
  ⟹ conf (aa, ba(| dyn_lenv := locals (aa, ba)(va ↦ v) —)) (G, L)

lemma vconf_upd_locals_rtclass:
  [|wf_prog G; (S, T') ∈ widen G; va ≠ This; va ≠ TContext; lenv L va = Some T'; T' ≠ NT;
  res_conf G (aa, ba(| dyn_lenv := dyn_lenv ba(va ↦ v) |)) (Inl v) (Inl S); conf (aa, ba) (G, L);
  dyn_lenv ba va = Some vb; vconf G aa (the_Addr (the (dyn_lenv ba TContext))) vb T'|]
  ⟹ vconf G aa (the_Addr (the (dyn_lenv ba TContext))) v T'

lemma res_conf_locals_lcl:
  [|wf_prog G; va ≠ This; va ≠ TContext; res_conf G (aa, ba) (Inl v) (Inl S)|]
  ⟹ res_conf G (aa, ba(| dyn_lenv := locals (aa, ba)(va ↦ v) |)) (Inl v) (Inl S)
```

Zugriff auf eine nicht leere Liste von Ausdrücken Die Konformität zwischen dynamischen Typen einer Liste von Werten von Termen und den statischen Typen dieser Terme ist unter anderem mit Hilfe des Lemmas tcontext_eq (siehe Abschnitt 9.1) gezeigt. Das Lemma tcontext_eq bringt zum Ausdruck, dass der Teamkontext eines Programmzustandes vor der Auswertung eines Terms und im Programmnachzustand gleich sind.

```
lemma res_conf_Cons:
  [|((None, ab, ba), Inr es, Inr vs, None, ad, bb) ∈ eval G;
  wf_prog G; (G, L) |= Inl (In2 e)::Inl Ta; (G, L) |= Inr es::Inr Ts;
  res_conf G (ab, ba) (Inl v) (Inl Ta); heap_ex ab ad; res_conf G (ad, bb) (Inr vs) (Inr Ts)|]
  ⟹ res_conf G (ad, bb) (Inr (v # vs)) (Inr (Ta # Ts))
```

9.3 Aussage über das *Confinement* von Rollenobjekten

Es wird eine Aussage über das *Confinement* von Rollenobjekten formuliert. Diese Aussage bringt zum Ausdruck, dass Rollenobjekte nur innerhalb des Kontextes der sie umschließenden Teaminstanz referenziert werden. Sie wird unter Verwendung der in Abschnitt 9.2 untersuchten Typzuverlässigkeitsaussage über das modellierte Typsystem für ObjectTeams/Java nachgewiesen. Diese ist im Rahmen dieser Arbeit weitgehend bewiesen (siehe Abschnitt 10.2 hinsichtlich einer Bewertung der noch offenen Beweisziele).

Zur Formalisierung der Aussage über das *Confinement* von Rollenobjekten werden folgende Prädikate definiert. Das Prädikat is_role_reference überprüft, ob es sich bei einem Wert v um eine Referenz auf ein Rollenobjekt handelt.

```
constdefs
  is_role_reference :: prog ⇒ aheap ⇒ val ⇒ bool
  is_role_reference G h v ==
    is_reference v & is_rclass G (obj_class (the (h (the_Addr v))))
```

Das Prädikat role_is_referenced_in_context überprüft - vorausgesetzt es handelt sich bei dem übergebenen Wert v um eine Referenz auf ein Rollenobjekt - ob ein Rollenobjekt innerhalb des Teamkontextes, in dem es enthalten ist, referenziert wird. Das heisst, es wird überprüft, ob die Identität des dynamischen Teamkontextes des dynamischen lokalen Environments gleich der Identität des Teamobjekts ist, in dem das Rollenobjekt enthalten ist, und auf das es einen Verweis speichert. Das Prädikat role_referenced_in_context dient zur einheitlichen Überprüfung eines Wertes und einer Liste von Werten.

```
consts
  role_is_referenced_in_context :: prog ⇒ state ⇒ val ⇒ bool
  role_referenced_in_context :: prog ⇒ state ⇒ vals ⇒ bool

defs
  role_is_referenced_in_context_def:
  role_is_referenced_in_context G s v ==
    (locals s) TContext ≠ None & tid (obj_dic (the ((heap s) (the_Addr v)))) ≠ None &
    id_TContext s = the (tid (obj_dic (the ((heap s) (the_Addr v))))))
```

```
role_referenced_in_context_def:
role_referenced_in_context G s vs ==
  (case vs of
    Inl v ⇒ is_role_reference G (heap s) v −→ role_is_referenced_in_context G s v
    | Inr vl ⇒ ∀ v ∈ set vl. is_role_reference G (heap s) v −→ role_is_referenced_in_context G s v)
```

Das Theorem confinement_of_role_objects über das *Confinement* von Rollenobjekten bringt folgendes zum Ausdruck. Wenn bei der Auswertung eines beliebigen Terms t relativ zu einem Programm G ein Resultatwert v produziert wird, das Programm wohlgeformt ist, der Programmzustand vor Auswertung des Terms t konform zum statischen Environment ist, ein Typ T für den Term t abgeleitet werden kann und bei der Auswertung des Terms t keine Exception aufgetreten ist, also x'=None gilt, dann gilt, dass wenn es sich bei dem produzierten Wert v um eine Referenz auf ein Rollenobjekt handelt, dass diese aus dem Kontext heraus auf das Rollenobjekt existiert, in dem das Rollenobjekt enthalten ist. Das bedeutet, das auf ein Rollenobjekt nur innerhalb des Kontextes zugegriffen werden kann, in dem es enthalten ist. Eine Teaminstanz stellt also eine inhärente Grenze für Referenzen auf Rollenobjekte dar.

Das Theorem confinement_of_role_objects über das *Confinement* von Rollenobjekten ist unter Verwendung der Typzuverlässigkeits- und der statischen Schichtenaussage, der Lemmata eval_Inj_elim und wt_Inj_elim und der Lemmata confinement_of_role_obj und confinement_of_role_objs nachgewiesen. Die Lemmata confinement_of_role_obj und confinement_of_role_objs bringen folgendes zum Ausdruck.

- Lemma confinement_of_role_obj: Wenn ein Programm wohlgeformt ist, der Programmzustand nach Auswertung des Terms t konform zum statischen Environment ist, es sich bei dem produzierten Resultatwert aa um eine Referenz auf ein Rollenobjekt handelt, der dynamische Typ dieser Referenz konform zum bei der Typisierung des Terms t abgeleiteten Typ a ist und dieser Typ wohlkonstruiert ist, dann wird das durch den Verweis aa referenzierte Rollenobjekt aus dem Kontext heraus, in dem es enthalten ist, referenziert.

  Diese Aussage ist per Fallunterscheidung über den Typ a des Terms t und Fallunterscheidungen über die statische und dynamische Weitungsbeziehung zwischen Typen bewiesen.

- Lemma confinement_of_role_objs: Für alle Listen von Typen T für die gilt, dass ein Programm G wohlgeformt ist, der Programmzustand nach Auswertung der Liste der Terme t konform zum statischen Environment ist, die dynamischen Typen der bei der Auswertung der Liste der Terme t produzierten Liste von Resultatwerten v konform zur Liste der statischen Typen T der Liste der Terme t ist, diese Liste von Typen wohlkonstruiert ist und für die gilt, dass es sich bei mindestens einem Wert der Liste von Resultatwerten um eine Referenz auf ein Rollenobjekt handelt, gilt, dass dieses Rollenobjekt aus dem Kontext heraus referenziert wird, in dem es enthalten ist.

  Diese Aussage ist per Induktion über die Liste der Resultatwerte v und unter Verwendung des Lemmas confinement_of_role_obj nachgewiesen.

```
theorem confinement_of_role_objects:
  [|G ⊢ (x,(h,l)) −t>> (v,x',(h',l')); wf_prog G; conf (h,l) (G,L); (G,L) |= t :: T|]
  ⟹ (x'=None −→ role_referenced_in_context G (h',l') v)

lemma confinement_of_role_obj:
  [|wf_prog G; conf (h', l') (G, L); is_role_reference G (heap (h', l')) aa;
  vconf G (heap (h', l')) (id_TContext (h', l')) aa a; is_wc_type (G, L) a|]
  ⟹ role_is_referenced_in_context G (h', l') aa

lemma confinement_of_role_objs [rule_format]:
  ∀ T. wf_prog G −→ conf (h',l') (G,L) −→
    list_all2 (vconf G (heap (h', l')) (id_TContext (h', l'))) v T −→
      wc_type (G, L) (Inr T) −→ is_role_reference G h' va −→ va ∈ set v −→
        role_is_referenced_in_context G (h', l') va
```

10 Zusammenfassung und Ausblick

Die Ergebnisse dieser Arbeit werden in Abschnitt 10.1 zusammengefasst und bewertet. In Abschnitt 10.2 wird dargestellt, welche Fragestellung sich bei der Analyse der Typsicherheit von ObjectTeams/Java ergeben hat, und welche Teile der Typzuverlässigkeitsaussage über das modellierte Typsystem für ObjectTeams/Java noch zu beweisen sind. Erfahrungen bei der Einbettung von ObjectTeams/Java in Isabelle/HOL werden in Abschnitt 10.3 beschrieben. Mögliche Erweiterungen der Modellierung und weiterführende Arbeiten werden in Abschnitt 10.4 vorgestellt.

10.1 Zusammenfassung und Bewertung des in dieser Arbeit Erreichten

Ein Typsystem für ObjectTeams/Java ist in Isabelle/HOL formalisiert und analysiert. Die Einbettung der Programmiersprache ObjectTeams/Java in Isabelle/HOL stellt eine semantisch formale Fundierung von ObjectTeams/Java dar.

Formalisierung des Team- und Rollenklassenkonzepts Die Konzepte von Team- und Rollenklassen sind als strukturelle Klassentypen modelliert. Dies ist unter anderem nötig, weil vereinfachend ein flacher Namensraum zur Modellierung von Typnamen verwendet wird, ermöglicht aber eine einfache Erweiterung der Modellierung zur Formalisierung von Bindungen von Rollenklassen an Klassen, die in von Teamklassen adaptierten Anwendungen enthalten sind (siehe Abschnitt 10.4).

Formalisierung der Vererbung zwischen Teamklassen Die Semantik der Vererbung zwischen Teamklassen mit einer Vererbung von Rollenklassen ist formalisiert.

Formalisierung des Zweifacherbens von Rollenklassen Das Erben von Rollenklassen entlang der `extends`- und der impliziten Vererbungshierarchie ist formalisiert.

Formalisierung eines Typsystems für ObjectTeams/Java mit einem Modell instanzbasierter Typen Ein Typsystem für ObjectTeams/Java mit einem Modell instanzbasierter Typen zur Darstellung der Abhängigkeit der Bindung von Rollentypen von dem Teamkontext, in dem sie vorkommen, ist modelliert und in Isabelle/HOL kodiert. Eine statische Schichtenaussage über die Wohlkonstruiertheit von bei der Typisierung von ObjectTeams/Java-Konstrukten abgeleiteten Typen einschließlich der Bindung von Rollentypen abhängig von dem Teamkontext, in dem sie vorkommen, ist nachgewiesen. Die formale Spezifikation von Regeln zur Typisierung von ObjectTeams/Java-Konstrukten einschließlich der darin enthaltenen Einschränkungen bezüglich der Wohlgetyptheit von ObjectTeams/Java-Konstrukten kann bei der Implementierung von ObjectTeams/Java-Compilern hilfreich sein.

Spezifikation von Wohlgeformtheitsbedingungen Eine Menge von Wohlgeformtheitsbedingungen ist spezifiziert. Sie entsprechen beim Kompilieren von ObjectTeams/Java-Programmen durchgeführten Überprüfungen. Die formale Spezifikation dieser Wohlgeformtheitsbedingungen kann als Richtlinie bei der Konstruktion von ObjectTeams/Java-Compilern verwendet werden.

Betrachtungen einer Typzuverlässigkeitsaussage des modellierten Typsystems für ObjectTeams/Java Eine Typzuverlässigkeitsaussage über das modellierte Typsystem für ObjectTeams/Java ist formuliert und in großen Teilen nachgewiesen. Sie impliziert, dass das Sprachdesign von ObjectTeams/Java in Kombination mit einer Menge von Wohlgeformtheitsbedingungen das Auftreten von unerwarteten Typfehlern bei der Ausführung von wohlgeformten ObjectTeams/Java-Programmen verhindert. Die Analyse der Typzuverlässigkeitsaussage über das modellierte Typsystem für ObjectTeams/Java hat unter anderem ergeben, dass die derzeitigen Regeln für die Bildung von Subteamklassen eine modulare Typüberprüfung von ObjectTeams/Java-Programmen brechen (siehe Abschnitt 10.2).

Analyse des *Confinement* von Rollenobjekten Die Kontrolle über Referenzen auf Rollenobjekte, die das Teams-Konstrukt ausübt, ist analysiert. Unter der Voraussetzung, dass die Typzuverlässigkeitsaussage über das modellierte Typsystem für ObjectTeams/Java vollständig nachgewiesen werden kann, ist gezeigt, dass Rollenobjekte nur innerhalb des Teamkontextes referenziert werden können, in dem sie enthalten

sind. Die Einschränkungen zur Realisierung von *confined role objects* sind spezifiziert und können als Richtlinie bei der Konstruktion von ObjectTeams/Java-Compilern verwendet werden.

10.2 Bewertung der im Rahmen dieser Arbeit offen gebliebenen Aspekte

Modulare Typüberprüfung von ObjectTeams/Java-Programmen Die Analyse der Typzuverlässigkeitsaussage über das modellierte Typsystem für ObjectTeams/Java hat unter anderem ergeben, dass für deren Nachweis eine Aussage über die modulare Typüberprüfung von ObjectTeams/Java-Programmen notwendig ist. Eine modulare Typüberprüfung von ObjectTeams/Java-Programmen bedeutet, dass die Wohlgeformtheit von Team- und Rollenklassendeklarationen nicht durch Subteamklassenbildungen beeinflusst wird. In Abschnitt 7.1.5 ist erläutert, an welcher Stelle die derzeitigen Regeln für die Bildung von Subteamklassen die Wohlgeformtheit von Superteam- und impliziten Superrollenklassendeklaration brechen können.

Es ist eine offene Fragestellung, ob die Regeln für die Bildung von Subteamklassen so verändert werden können, dass eine modulare Typüberprüfung von ObjectTeams/Java-Programmen möglich ist, oder ob bei der Bildung einer Subteamklasse auch alle direkten und indirekten Superteam- und impliziten Superrollenklassendeklarationen hinsichtlich ihrer Wohlgeformtheit im Kontext dieser Subteamklasse überprüft werden müssen, um eine Aussage über die Wohlgeformtheit von Superteam- und impliziten Superrollenklassendeklarationen im Kontext dieser Subteamklasse treffen zu können.

Offene Aspekte bei der Betrachtung der Typzuverlässigkeitsaussage über das modellierte Typsystem für ObjectTeams/Java Im Rahmen dieser Arbeit sind folgende Teile der Typzuverlässigkeitsaussage über das modellierte Typsystem für ObjectTeams/Java nicht komplett nachgewiesen, sondern mittels Axiomen formuliert und konzeptionell analysiert.

Instanziierung einer Klasse Zum Nachweis der Typzuverlässigkeit des Typsystems für ObjectTeams/Java muss hinsichtlich der Instanziierung einer Klasse gezeigt werden, dass der um das neu erzeugte Objekt aktualisierte Programmzustand konform zum statischen Environment ist. Die Objektkonformität des neu erzeugten Objekts muss nachgewiesen werden. Die Gleichheit der bei der statischen Analyse und der während der Ausführung ermittelten zu instanziierenden Rollenklasse muss gezeigt werden. Diese Nachweise sollten aufgrund der Tatsache, dass die Konformität zwischen dynamischen Typen von Default-Werten von Attributen eines Objekts und den statischen Typen dieser Attribute betrachtet werden muss, und dass Rollenklassen innerhalb eines Teamkontextes eindeutig sind, zu erbringen sein und sind im Rahmen dieser Arbeit aus Zeitgründen nicht durchgeführt.

Attributzugriff Zum Nachweis der Typzuverlässigkeit des Typsystems für ObjectTeams/Java wird hinsichtlich eines Attributzugriffs eine Aussage über die Wohlgeformtheit von entlang einer Teamklassenvererbungshierarchie geerbten Attributen benötigt (siehe Abschnitt 7.1.5). Die innerhalb dieser Aussage zu behandelnde Vererbung von Attributen an Rollenklassen entlang der `extends`- und der impliziten Vererbungshierarchie stellt eine Herausforderung dar. Statisch sind Einschränkungen formuliert, die sicherstellen sollten, dass der dynamische Typ des bei der Auswertung eines Attributzugriffsausdrucks produzierten Wertes konform zum statischen Typ dieses Attributzugriffsausdrucks ist.

Attributzuweisung Zum Nachweis der Typzuverlässigkeit des Typsystems für ObjectTeams/Java werden hinsichtlich einer Attributzuweisung zum einen dieselben Aussagen wie bei der Betrachtung eines Attributzugriffsausdrucks im Rahmen der Analyse der Typzuverlässigkeit dieses Typsystems benötigt. Zum anderen kann die statisch geforderte Einschränkung hinsichtlich der Subtypbeziehung zwischen dem Typ des zugewiesenen Ausdrucks und dem Typ des Attributs innerhalb dieses Nachweises verwendet werden.

Methodenaufruf Der Nachweis der Typzuverlässigkeit des Typsystems für ObjectTeams/Java hinsichtlich eines Methodenaufrufs erfordert zum einen eine Aussage über die Wohlgeformtheit von geerbten Methoden entlang einer Teamklassenvererbungshierarchie. Zum anderen sind drei Axiome hinsichtlich der Konformität zwischen dem dynamischen Typ des bei der Auswertung des Methodenaufrufausdrucks produzierten Wertes und dem während der statischen Analyse ermittelten Typ dieses Methodenaufrufausdrucks, der Konformität zwischen dem während des Methodenaufrufs aktualisierten Programmzustands

und dem aktualisierten statischen Environment und zwischen dem aktualisierten Programmnachzustand und dem statischen Environment im Rahmen dieser Arbeit aus Zeitgründen nicht erbracht. Die im statischen Teil formulierten Einschränkungen beim Aufruf einer Methode sollten sicherstellen, dass diese Konformitätsnachweise erbracht werden können.

10.3 Erfahrungen bei der Einbettung von ObjectTeams/Java in Isabelle/HOL

Obwohl die Erfahrungen mit interaktivem Theorembeweisen zu Beginn dieser Arbeit gering waren, war die Verwendung des Beweiswerkzeugs Isabelle/HOL bei der Formalisierung eines Typsystems für ObjectTeams/Java und beim Nachweisen von Aussagen über diese Modellierung aus folgenden Gründen hilfreich. Typfehler innerhalb von Definitionen werden bei der Kodierung automatisch erkannt. Inkonsistenzen in Definitionen und Beweisen, die insbesondere bei Veränderungen einer Modellierung entstehen, werden automatisch aufgedeckt. Die Durchführung von Beweisen bis ins Detail deckt fehlende Beweisschritte auf.

Insbesondere der *Simplifier* - ein in Isabelle integriertes Beweistool zur Vereinfachung von Beweiszielen durch wiederholte Anwendung von Gleichungen - konnte als maschinelle Unterstützung beim Durchführen von Beweisen genutzt werden.

Die Formalisierung von Java namens μJava [KNO+02a], an die diese Arbeit unter anderem angelehnt ist, hat durch ihre Verständlichkeit, modulare Gestaltung und Erweiterbarkeit überzeugt. Viele Konzepte zur Modellierung der Java-Bestandteile von ObjectTeams/Java sind an die in [KNO+02a] und [KNO+02b] enthaltenen Formalisierungen angelehnt. Letztere ist durch ihren Umfang aufgrund der Formalisierung eines bedeutend größeren Sprachumfangs von Java schwerer zugänglich.

10.4 Weiterführende Arbeiten

Formalisierung weiterer Bestandteile des Programmiermodells Object Teams Die Gesichtspunkte des Programmiermodells Object Teams zur Bindung von Rollenklassen an Klassen von durch Teamklassen adaptierten Anwendungen könnten durch Erweiterungen der in dieser Arbeit vorgestellten Einbettung von ObjectTeams/Java in Isabelle/HOL formalisiert werden. Diese würden unter anderem eine Erweiterung des Rollenklassentyps zur Formalisierung von Klassen- und Methodenbindungen erforderlich machen. Das Schlüsselwort `base` zum Zugriff auf ersetzte Basismethoden müsste formalisiert werden. Desweiteren müsste das Konzept des *translation polymorphism* zur Definition der Konformität zwischen Rollen- und gebundenen Klassen formalisiert werden. Die Typüberprüfungen, Wohlgeformtheitsüberprüfungen und die Semantik von ObjectTeams/Java müssten zur Formalisierung von Methodenbindungen und der Aktivierung und Deaktivierung von Teamobjekten zur Adaptierung von Anwendungen erweitert werden. Die Erweiterung der Modellierung um diese Gesichtspunkte von Object Teams ist insbesondere hinsichtlich der Formalisierung der aspektorientierten Bestandteile von Object Teams interessant. Die Erweiterbarkeit der in dieser Arbeit vorgestellten Modellierung würde in diesem Rahmen untersucht werden.

Formalisierung des Konzepts von anonymen Methoden zur Wiederverwendung von aus Basisklassen geerbten Methoden in Rollenklassen Das Konzept von anonymen Methoden wird in [BV99] zur verstärkten Kontrolle von Referenzen auf Objekte vorgestellt. Eine Methode ist anonym, wenn sie die Referenz auf das *target* des Methodenaufrufs nicht enthüllt. Es werden eine Menge von Einschränkungen formuliert, die eine Methode erfüllen muss, um anonym zu sein. Anonyme Methoden ermöglichen die Wiederverwendung von Methoden in einem Kontext, in dem bestimmte Objekte nicht beliebig referenziert werden sollen.

Die Formalisierung des Konzeptes von anonymen Methoden ist im Kontext der Betrachtung von *confined role objects* interessant, weil dadurch eine Menge von aus Basisklassen geerbten Methoden identifiziert werden könnte, die auf Rollenobjekten aufgerufen und wiederverwendet werden können. Die Kontrolle von Referenzen auf Rollenobjekte, die das Teams-Konstrukt ausübt, bleibt dabei erhalten, weil eine anonyme Methode die Referenz auf das Objekt, auf dem sie ausgeführt wird, nicht enthüllt.

Zur Realisierung des Konzeptes von anonymen Methoden müssten die in [BV99] spezifizierten Kriterien zur Feststellung, ob eine Methode anonym ist, bei Bestimmung der Menge aller in einem Programm enthaltenen anonymen Methoden während der Typisierung dieses Programms überprüft werden. Die Einschränkung, welche Methoden auf einem Rollenobjekt aufgerufen werden dürfen, müsste hinsichtlich der Zulassung des Aufrufs von aus Superbasisklassen geerbten anonymen Methoden erweitert werden.

Innerhalb des Prädikats zur Definition der Konformität zwischen einem dynamischen Typ eines Wertes eines Terms und dem statischen Typ des Terms müsste die Weitung von Rollentypen hin zu Basistypen erlaubt werden, wenn der Term in einer anonymen Methode enthalten ist.

Formalisierung von *externalized roles* *Externalized roles* stellen Rollenobjekte dar, die uneingeschränkt sichtbar sind. Ihre Typen sind relativ zu Bezeichnern von finalen Variablen verankert, die Referenzen auf die Teamobjekte speichern, in denen sie enthalten sind. Zur Formalisierung von *externalized roles* müsste das Java-Konzept von finalen Variablen [GJS+00] (§4.5.4) formalisiert werden. Der Typ zur Modellierung des Instanzkontextes von statischen instanzbasierten Typen müsste um einen Konstruktor zur Formalisierung von Bezeichnern als Instanzkontext erweitert werden. Die Typrelationen wären zu erweitern, und bei der Typisierung von Programmen müsste eine Flußanalyse zur Überprüfung, ob zwei *externalized roles* in derselben Teaminstanz enthalten sind, formalisert werden. In [Ern01] wird die Ansicht vertreten, dass diese nicht Teil einer Typüberprüfung sein sollte. Alternativ könnte versucht werden, das Modell der instanzbasierten Typen durch ein abstrakteres formales Konstrukt zu repräsentieren. Das im Rahmen dieser Arbeit formalisierte und analysierte Typsystem für ObjectTeams/Java realisiert ein Konzept einer instanzbasierten Typisierung von ObjectTeams/Java-Konstrukten und kann zur Formalisierung von *externalized roles* erweitert werden oder als Basis für die Erarbeitung eines abstrakteren formalen Konstrukts zur Repräsentation von instanzbasierten Typen dienen.

Literatur

[BV99] B. Bokowski, J. Vitek, Confined Types, Proceedings 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99), 1999.

[Car97] L. Cardelli, Type Systems, Handbook of Computer Science and Engineering, Chapter 103, CRC Press, 1997.

[Chu40] A. Church, A Formulation of the Simple Theory of Types, Journal of Symbolic Logic, pp. 56-68, 1940.

[Ern01] E. Ernst, Family Polymorphism, In Proceedings ECOOP 2001, LNCS 2072, Springer-Verlag, pp. 303-326, 2001.

[GJS+00] J. Gosling, B. Joy, G. Steele, G. Bracha, The Java Language Specification, Second Edition, 2000, http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html, 12.09.2003.

[Her03a] S. Herrmann, Object Teams: Improving Modularity for Crosscutting Collaborations, In Proceedings „Objects, Components, Architectures, Services, and Applications for a Networked World", LNCS 2591, Springer-Verlag, pp. 248-264, 2003.

[Her03b] S. Herrmann, Sustainable Architectures by Combining Flexibility and Strictness in Object Teams, zur Veröffentlichung eingereicht, 2003.

[Her03c] S. Herrmann, ObjectTeams/Java Language Definition, http://www.objectteams.org, 12.09.2003.

[KNO+02a] G. Klein, T. Nipkow, D. von Oheimb, C. Pusch, M. Strecker, Java Source and Bytecode Formalizations in Isabelle: μJava, 2002, http://isabelle.in.tum.de/dist/past.html, 12.09.2003.

[KNO+02b] G. Klein, T. Nipkow, D. von Oheimb, L. P. Nieto, N. Schirmer, M. Strecker, Java Source and Bytecode Formalizations in Isabelle: Bali, 2002, http://isabelle.in.tum.de/dist/past.html, 12.09.2003.

[NOP99] T. Nipkow, D. von Oheimb, C. Pusch, μJava: Embedding a Programming Language in a Theorem Prover, In Proceedings of the International Summer School Marktoberdorf, 1999.

[NPW02] T. Nipkow, L. C. Paulson, M. Wenzel, A Proof Assistant for Higher-Order Logic, Springer-Verlag, 2002.

[Ohe01] D. von Oheimb, Analyzing Java in Isabelle/HOL, Ph.D. thesis, TU München, 2001.

[ON99] D. von Oheimb, T. Nipkow, Machine-Checking the Java Language Specification: Proving Type-Safety, In Jim Alves-Foss (Ed): Formal Syntax and Semantics of Java, LNCS 1523, pp. 119-156, Springer-Verlag, 1999.

[Pau94] L. C. Paulson, *Isabelle: A Generic Theorem Prover*, LNCS 828, Springer-Verlag, 1994.

[Pau02a] L. C. Paulson, The Isabelle Reference Manual, 2002, http://isabelle.in.tum.de/dist/past.html, 12.09.2003.

[Pau+02b] L. C. Paulson et al., Isabelle - Higher Order Logic, 2002, http://isabelle.in.tum.de/dist/past.html, 12.09.2003.

[Pep97] P. Pepper, Programmiersprachen und -systeme, 1997, http://flp.cs.tu-berlin.de/lehre/sose03/pss/index.html, 12.09.2003.

[Szy98] C. Szyperski, Component Software, Addison-Wesley, 1998.

[Wen02] M. Wenzel, The Isabelle/Isar Reference Manual, 2002, http://isabelle.in.tum.de/dist/past.html, 12.09.2003.

Anhang

header {* BoardGame - ObjectTeams/Java Example *}

theory BoardGame = Decl:

```
(*
abstract team class BoardGame {
   Board board;
   Rule rule1;
   void initialize() {
     this.board = new Board();
     this.board.initialize();
     this.addRules();
   }
   abstract void addRules();
   boolean checkConsistencyWithRules(Player player, int x, int y,
                                     Token token) {
     boolean ok = false;
     if (this.rule1.trigger(player,x,y,token)) {
       ok = true;
     } else {
       ;
     }
     return ok;
   }
   abstract class Board {
     abstract void initialize();
     boolean isFieldFree(int x, int y) { return true; }
   }
   abstract class Rule {
     abstract boolean trigger(Player player, int x, int y, Token token);
   }
   class Player {}
   class Token {}
}
team class TicTacToe extends BoardGame {
   void addRules () { this.rule1 = new RuleFreeField(); }
   class Board {
     void initialize() {}
   }
   class Player {}
   class Rule {
     boolean trigger(Player player, int x, int y, Token token) { return true; }
   }
   class RuleFreeField extends Rule {
     boolean trigger(Player player, int x, int y, Token token) {
       boolean ok = false;
       if (super.trigger(player,x,y,token)) }
         if (TicTacToe.this.board.isFieldFree(x,y)) {
           ok = true;
         } else {
           ok = false;
         }
       } else {
         ok = false;
       }
```

```
      return ok;
    }
  }
}
*)

section {* Type, Variable and Method Names *}

datatype tnam_
  = BoardGame_
  | TicTacToe_
  | BoardGame_Board_
  | BoardGame_Player_
  | BoardGame_Rule_
  | BoardGame_Token_
  | TicTacToe_Board_
  | TicTacToe_Player_
  | TicTacToe_Rule_
  | TicTacToe_RuleFreeField_
  | TicTacToe_Token_

datatype mname_
  = initialize_
  | addRules_
  | checkConsistencyWithRules_
  | isFieldFree_
  | trigger_

datatype vnam_
  = rule1_
  | board_
  | player_
  | x_
  | y_
  | token_
  | ok_

consts
  tnam_ :: tnam_ ⇒ tnam
  mname_ :: mname_ ⇒ mname
  vnam_ :: vnam_ ⇒ vnam

(* Aus Bali:
tnam_, vnam_ and mname_ are intended to be isomorphic
to tnam, vnam and mname *)

axioms
  inj_tnam_ [simp]: (tnam_ x = tnam_ y) = (x = y)
  inj_mname_ [simp]: (mname_ x = mname_ y) = (x = y)
  inj_vnam_ [simp]: (vnam_ x = vnam_ y) = (x = y)

axioms
  surj_tnam_: EX m. n = tnam_ m
  surj_vnam_: EX m. n = vnam_ m
  surj_mname_: EX m. n = mname_ m

syntax
  BoardGame :: tname
  TicTacToe :: tname
  BoardGame_Board :: tname
  BoardGame_Player :: tname
```

BoardGame_Rule :: $tname$
BoardGame_Token :: $tname$
TicTacToe_Board :: $tname$
TicTacToe_Player :: $tname$
TicTacToe_Rule :: $tname$
TicTacToe_RuleFreeField :: $tname$
TicTacToe_Token :: $tname$
initialize :: $mname$
addRules :: $mname$
checkConsistencyWithRules :: $mname$
isFieldFree :: $mname$
trigger :: $mname$
rule1 :: $vname$
board :: $vname$
ok :: $vname$
player :: $vname$
x :: $vname$
y :: $vname$
token :: $vname$

`translations`

BoardGame == TName (tnam_ BoardGame_)
TicTacToe == TName (tnam_ TicTacToe_)
BoardGame_Board == TName (tnam_ BoardGame_Board_)
BoardGame_Player == TName (tnam_ BoardGame_Player_)
BoardGame_Rule == TName (tnam_ BoardGame_Rule_)
BoardGame_Token == TName (tnam_ BoardGame_Token_)
TicTacToe_Board == TName (tnam_ TicTacToe_Board_)
TicTacToe_Player == TName (tnam_ TicTacToe_Player_)
TicTacToe_Rule == TName (tnam_ TicTacToe_Rule_)
TicTacToe_RuleFreeField == TName (tnam_ TicTacToe_RuleFreeField_)
TicTacToe_Token == TName (tnam_ TicTacToe_Token_)
initialize == mname_ initialize_
addRules == mname_ addRules_
checkConsistencyWithRules == mname_ checkConsistencyWithRules_
isFieldFree == mname_ isFieldFree_
trigger == mname_ trigger_
rule1 == VName (vnam_ rule1_)
board == VName (vnam_ board_)
ok == VName (vnam_ ok_)
player == VName (vnam_ player_)
x == VName (vnam_ x_)
y == VName (vnam_ y_)
token == VName (vnam_ token_)

section {* Method Declarations *}

subsection {* Method Declaration of Standard Classes *}

`defs`

Object_mdecls_def:
Object_mdecls == []

Team_mdecls_def:
Team_mdecls == []

NullPointer_mdecls_def:
NullPointer_mdecls == []

ClassCast_mdecls_def:

```
  ClassCast_mdecls == []

  OutOfMemory_mdecls_def:
  OutOfMemory_mdecls == []

subsection {* Methods Declarations of BoardGame *}

constdefs
  initialize_BG_mdecl :: mdecl
  initialize_BG_mdecl ==
    ((|name = initialize, param_tys = []|),
     (|abstract = False, static = False, res_ty = PrimT Void, params = [],
       body= Some (|lcl_vars = [],
         stmt = (Expr ({BoardGame}this..board:=NewC TThis BoardGame_Board));;
                  ((Expr ({BoardGame,IntVir}TargetExpr ({BoardGame}this..board)..initialize([])));;
                   (Expr ({BoardGame,IntVir}TargetExpr this..addRules([])))),
         res = Lit Unit|)|))

  addRules_BG_mdecl :: mdecl
  addRules_BG_mdecl ==
    ((|name = addRules, param_tys = []|),
     (|abstract = True, static = False, res_ty = PrimT Void, params = [],
       body = None|))

  checkConsistencyWithRules_mdecl :: mdecl
  checkConsistencyWithRules_mdecl ==
    ((|name = checkConsistencyWithRules,
       param_tys = [Class TThis BoardGame_Player, PrimT Integer,
                    PrimT Integer, Class TThis BoardGame_Token]|),
     (|abstract = False, static = False, res_ty = PrimT Boolean,
       params = [player, x, y, token],
       body = Some (|lcl_vars = [(ok, PrimT Boolean)],
         stmt = Comp
                  (Expr (LAss ok (Lit (Bool False))))
                   (Cond (Call BoardGame IntVir
                     (TargetExpr (FAcc BoardGame (LAcc This) rule1))
                       trigger [(LAcc player), (LAcc x), (LAcc y), (LAcc token)])
                  (Expr (LAss ok (Lit (Bool True)))) Skip),
         res = LAcc ok|)|))

  initialize_BG_Board_mdecl :: mdecl
  initialize_BG_Board_mdecl ==
    ((|name = initialize, param_tys = []|),
     (|abstract = True, static = False, res_ty = PrimT Void, params = [],
       body = None|))

  isFieldFree_mdecl :: mdecl
  isFieldFree_mdecl ==
    ((|name = isFieldFree, param_tys = [PrimT Integer, PrimT Integer]|),
     (|abstract = False, static = False, res_ty = PrimT Boolean, params = [x, y],
       body = Some (|lcl_vars = [],
         stmt = Skip,
         res = Lit (Bool True)|)|))

  trigger_BG_mdecl :: mdecl
  trigger_BG_mdecl ==
    ((|name = trigger,
      param_tys = [Class TThis BoardGame_Player, PrimT Integer,
                   PrimT Integer, Class TThis BoardGame_Token]|),
     (|abstract = True, static = False, res_ty = PrimT Boolean,
       params = [player,x,y,token],
```

```
        body = None|))

subsection ∗ Method Declarations of TicTacToe ∗

constdefs

    addRules_TTT_mdecl :: mdecl
    addRules_TTT_mdecl ==
      ((|name = addRules, param_tys = []|),
       (|abstract = False, static = False, res_ty = PrimT Void, params = [],
        body = Some (|lcl_vars = [],
          stmt = Expr (FAss BoardGame (LAcc This) rule1
                    (NewC TThis TicTacToe_RuleFreeField)),
          res = Lit Unit|)|))

    initialize_TTT_Board_mdecl :: mdecl
    initialize_TTT_Board_mdecl ==
      ((|name = initialize, param_tys = []|),
       (|abstract = False, static = False, res_ty = PrimT Void, params = [],
        body = Some (|lcl_vars = [],
          stmt = Skip,
          res = Lit Unit|)|))

    trigger_TTT_mdecl :: mdecl
    trigger_TTT_mdecl ==
      ((|name = trigger,
        param_tys = [Class TThis BoardGame_Player, PrimT Integer,
                     PrimT Integer, Class TThis BoardGame_Token]|),
       (|abstract = False, static = False, res_ty = PrimT Boolean,
        params = [player,x,y,token],
        body = Some (|lcl_vars = [],
          stmt = Skip,
          res = Lit (Bool True)|)|))

    trigger_TTT_Sub_mdecl :: mdecl
    trigger_TTT_Sub_mdecl ==
      ((|name = trigger,
        param_tys = [Class TThis BoardGame_Player, PrimT Integer,
                     PrimT Integer, Class TThis BoardGame_Token]|),
       (|abstract = False, static = False, res_ty = PrimT Boolean,
        params = [player,x,y,token],
        body = Some (|lcl_vars = [(ok, PrimT Boolean)],
          stmt = Comp
                   (Expr (LAss ok (Lit (Bool False))))
                    (Cond (Call TicTacToe_Rule SuperM Super trigger
                             [LAcc player, LAcc x, LAcc y, LAcc token])
                      (Cond (Call BoardGame_Board IntVir (TargetExpr Tthis)
                               isFieldFree [LAcc x, LAcc y])
                        (Expr (LAss ok (Lit (Bool True))))
                        (Expr (LAss ok (Lit (Bool False)))))
                      (Expr (LAss ok (Lit (Bool False))))),
          res = LAcc ok|)|))

section {∗ Class Declarations ∗}

subsection {∗ Class Declarations BoardGame ∗}

constdefs
    BoardGame_decl :: tclass
    BoardGame_decl ==
      (|abstract = True,
```

```
      attrs = [(rule1, (|field_ty = Class TThis BoardGame_Rule|)),
                    (board, (|field_ty = Class TThis BoardGame_Board|))],
      methods = [initialize_BG_mdecl, addRules_BG_mdecl,
      checkConsistencyWithRules_mdecl],
      super = Team,
      roles = [BoardGame_Board, BoardGame_Player, BoardGame_Rule,
                    BoardGame_Token]|)

  BoardGame_Board_decl :: rclass
  BoardGame_Board_decl ==
    (|abstract = True,
      attrs = [],
      methods = [initialize_BG_Board_mdecl, isFieldFree_mdecl],
      super = Object,
      implicit = None,
      encl_class = BoardGame|)

  BoardGame_Player_decl :: rclass
  BoardGame_Player_decl ==
    (|abstract = False,
      attrs = [],
      methods = [],
      super = Object,
      implicit = None,
      encl_class = BoardGame|)

  BoardGame_Rule_decl :: rclass
  BoardGame_Rule_decl ==
    (|abstract = True,
      attrs = [],
      methods = [trigger_BG_mdecl],
      super = Object,
      implicit = None,
      encl_class = BoardGame|)

  BoardGame_Token_decl :: rclass
  BoardGame_Token_decl ==
    (|abstract = False,
      attrs = [],
      methods = [],
      super = Object,
      implicit = None,
      encl_class = BoardGame|)

subsection {* Class Declarations TicTacToe *}

constdefs
  TicTacToe_decl :: tclass
  TicTacToe_decl ==
    (|abstract = False,
      attrs = [],
      methods = [addRules_TTT_mdecl],
      super = BoardGame,
      roles = [TicTacToe_Board, TicTacToe_Player, TicTacToe_Rule,
      TicTacToe_RuleFreeField, TicTacToe_Token]|)

  TicTacToe_Board_decl :: rclass
  TicTacToe_Board_decl ==
    (|abstract = False,
      attrs = [],
```

```
      methods = [initialize_TTT_Board_mdecl],
      super = Object,
      implicit = Some BoardGame_Board,
      encl_class = TicTacToe|)

  TicTacToe_Player_decl :: rclass
  TicTacToe_Player_decl ==
    (|abstract = False,
      attrs = [],
      methods = [],
      super = Object,
      implicit = Some BoardGame_Player,
      encl_class = TicTacToe|)

  TicTacToe_Rule_decl :: rclass
  TicTacToe_Rule_decl ==
    (|abstract = False,
      attrs = [],
      methods = [trigger_TTT_mdecl],
      super = Object,
      implicit = Some BoardGame_Rule,
      encl_class = TicTacToe|)

  TicTacToe_RuleFreeField_decl :: rclass
  TicTacToe_RuleFreeField_decl ==
    (|abstract = False,
      attrs = [],
      methods = [trigger_TTT_Sub_mdecl],
      super = TicTacToe_Rule,
      implicit = None,
      encl_class = TicTacToe|)

  TicTacToe_Token_decl :: rclass
  TicTacToe_Token_decl ==
    (|abstract = False,
      attrs = [],
      methods = [],
      super = Object,
      implicit = Some BoardGame_Token,
      encl_class = TicTacToe|)

subsection {* Classes *}

constdefs
  boardgame_prog :: prog
  boardgame_prog ==
    ([(Object, ObjectC), (SXcpt NullPointer, NullPointerC),
     (SXcpt ClassCast, ClassCastC), (SXcpt OutOfMemory, OutOfMemoryC)],
     [(Team, TeamC), (BoardGame, BoardGame_decl), (TicTacToe, TicTacToe_decl)],
     [(BoardGame_Board, BoardGame_Board_decl),
     (BoardGame_Player, BoardGame_Player_decl),
     (BoardGame_Rule, BoardGame_Rule_decl), (BoardGame_Token, BoardGame_Token_decl),
     (TicTacToe_Board, TicTacToe_Board_decl), (TicTacToe_Player, TicTacToe_Player_decl),
     (TicTacToe_Rule, TicTacToe_Rule_decl),
     (TicTacToe_RuleFreeField, TicTacToe_RuleFreeField_decl),
     (TicTacToe_Token, TicTacToe_Token_decl)])


end
```