Hier könnte Ihr Unilogo stehen

Masterarbeit

Prinzipale Inhabitation im einfach getypten Lambda-Kalkül

Christoph Stahl
12. Oktober 2018

Gutachter:
Prof. Dr. Jakob Rehof
M. Sc. Jan Bessai

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl für Software Engineering (LS-14)
https://ls14-www.cs.tu-dortmund.de/

Inhaltsverzeichnis

1 Einleitung

Aus der Softwareentwicklung wissen wir, dass Programmierer nicht unfehlbar sind. Ein wichtiges Werkzeug, das viele Sprachen bereitstellen, um Fehler zu vermeiden, ist die Typisierung. Variablen und Funktionen können so festen Typen zugewiesen werden, die einen Hinweis auf die Verwendung geben. Durch statische Analyse des Quelltextes kann dann überprüft werden, ob diese Hinweise beachtet werden. Betrachten wir hier die statisch typisierte Sprache Java mit dem folgenden Codefragment.

```
public class Add {
  public static int add2(int i) {
      return i + 2;
  }
}
```

Die Funktion `Add.add2` ist somit eine Funktion, die einen ganzzahligen Wert als Eingabe erwartet und einen ganzzahligen Wert als Rückgabe hat. Über das Typsystem von Java kann nun entschieden werden, dass `Add.add2(2)` einen gültigen, aber `Add.add2("Esel")` einen ungültigen Aufruf an `Add.add2` darstellt. Insbesondere wissen wir, dass `Add.add2(2)` selbst wieder eine ganze Zahl ist und infolgedessen `Add.add2(Add.add2(2))` ein gültiger Aufruf ist. Offensichtlich ist `Add.add2("Esel")` ein Programmierfehler. Für trivialen Fälle, wie der hier präsentierte, lässt sich dieser Fehler auch sehr schnell von Hand finden und würde in echten Softwareprojekten so natürlich nicht auftreten. Echte Softwareprojekte sind jedoch deutlich komplexer. Falls der String `"Esel"` nicht als Stringliteral, sondern über die Rückgabe einer anderen Funktion an `Add.add2` übergeben wird, lässt sich dies nicht mehr so einfach nachvollziehen. Der falsche Aufruf könnte hier beispielsweise durch eine Änderung der Schnittstelle oder durch die Verwechslung der entsprechenden Funktion auftreten. Je komplexer eine Software wird, desto höher ist die Wahrscheinlichkeit, dass solche Fehler auftreten. Typisierte Sprachen helfen diese Fehler bereits zur Compilezeit zu entdecken und verhindern so fehlerhaftes Verhalten in der Ausführung.

In dieser Arbeit werden wir uns mit dem theoretischen Konstrukt des *einfach getypten Lambda-Kalküls* befassen. Dieser ist eine syntaktisch einfache Sprache zusammen mit

einem einfachen Typsystem. Trotz dieser Einfachheit übertragen sich viele Ergebnisse, die im einfach getypten Lambda-Kalkül erforscht wurden, auf komplexere Typsysteme. Wir können somit den einfach getypten Lambda-Kalkül als Kern für diese komplexeren Typsysteme betrachten. Gerade wegen der Einfachheit können viele theoretische Aussagen einfacher getroffen und bewiesen werden. Der einfach getypte Lambda-Kalkül agiert nicht nur als Kern vieler Typsysteme, er entspricht sogar genau einem Fragment der Logik. Es gilt, dass Typen Aussagen entsprechen, und Terme, die zu diesen Typen passen, den Beweisen für diese Aussage entsprechen.[1]

Wir werden in dieser Arbeit den Begriff des prinzipalen Typs betrachten. Der prinzipale Typ eines Terms ist der allgemeinste Typ eines Terms. Um eine Vorstellung zu bekommen, was ein allgemeinster Typ ist, betrachten wir erneut die Sprache Java, deren Typsystem Typvariablen über Generics unterstützt. Hierfür betrachten wir das folgende Codefragment.

```
public class Id {
    public static int id(int i) {
        return i;
    }
}
```

Es ist klar, dass `Id.id` genau den Wert zurückgibt, den die Methode übergeben bekommt. Die hier angegebene Definition der Identität ist jedoch beschränkt auf `int`. Derselbe Methodenrumpf kann jedoch für die Identität beliebiger Typen verwendet werden. Mittels Generics können wir einen Typ für die Methode angeben, der nicht auf `int` beschränkt ist.

```
public class Id_T {
  public static <T> T id(T i) {
    return i;
  }
}
```

Die Methode `Id_T.id` typt hier den Term `return i;` mit seinem allgemeinsten Typen, `T` $\rightarrow$ `T`.

Eine wichtige Fragestellung im Umgang mit Typsystemen ist die der Inhabitation. Die Frage nach der Inhabitation ist die Frage, ob es zu einem gegebenen Typen einen

[1] Dies werden wir im Rahmen dieser Arbeit auch betrachten.

Term gibt, der erfolgreich mit dem gegebenen Typen getypt werden kann. Diese Frage ist keineswegs trivial. Gehen wir davon aus, dass wir eine generische Typvariable `T` haben. Um `T` zu inhabitieren, müssten wir eine Funktion schreiben, die ohne Eingabe, jeden beliebigen Typen hat. Dies ist nicht ohne weiteres möglich[2].

Die Inhabitation ist in der Hinsicht interessant, da aus einem Algorithmus, der die Inhabitation entscheidet, auf einfache Art und Weise genau der Term extrahiert werden kann, der den Typ inhabitiert. Wenn über ein geeignetes Typsystem nun mittels Typen eine Spezifikation für Programme kodiert wird, lassen sich somit ganze Programme synthetisieren[4]. Des Weiteren lassen sich über den Zusammenhang zur Logik Aussagen über die Inhabitation von Typen, auf die Beweisbarkeit von logischen Aussagen übertragen.

Von besonderen Interesse ist hier auch die Frage der prinzipalen Inhabitation. Während die Frage der Inhabitation der Existenz eines Terms zu einem gegebenen Typen entspricht, entspricht die prinzipale Inhabitation der Existenz eins Terms, sodass der gegebene Typ der allgemeinste Typ des Terms ist. Übertragen auf die Synthese von Programmen entspricht dies einem Programm, dass genauestens einer Spezifikation entspricht; übertragen auf die Logik entspricht es einem Beweis, der nicht mehr beweist, als die gesuchte Aussage.

[12] zeigt, dass die Frage nach der prinzipalen Inhabitation weder deutlich schwieriger, noch deutlich einfacher als die der normalen Inhabitation zu beantworten ist. Beide Probleme sind PSPACE-vollständig. Das Enthaltensein in PSPACE folgt aus einem in [12] aufgestellten Algorithmus, der zum einen die prinzipale Inhabitation entscheidet und zum anderen die entsprechende Platzschranke einhält. Die PSPACE-Schwierigkeit ist in [12] mit einer Reduktion von dem ebenfalls PSPACE-vollständigen Problem der normalen Inhabitation, eingeschränkt auf eine bestimmte Art von Typen, bewiesen. Hierfür wird aus einem inhabitierten Typ τ ein prinzipal inhabitierter Typ τ^* erstellt, indem dieser mit Teiltypen von sich selber erweitert wird. Die genaue Konstruktion ist an dieser Stelle zunächst nebensächlich, relevant ist hier, dass der resultierende Typ τ^* stark von dem Ursprungstyp τ abhängt. Die Arbeit stellt des Weiteren die Vermutung auf, dass eine sehr ähnliche Konstruktion für Typen, die keiner Einschränkung unterliegen, möglich ist.

[2] Tatsächlich kann in Java einfach `null` zurückgegeben werden, da `null` jeden Typ inhabitiert. In dieser theoretischen Behandlung betrachten wir jedoch diese und ähnliche Fälle nicht, in denen beliebige Typen inhabitiert werden können. Des Weiteren haben bestimmte Java- und Scalaversionen einen Fehler, der es erlaubt das Typsystem zu überlisten und beliebige Typen zu inhabitieren.[2]

Das Beantworten der Vermutung ist nicht Teil dieser Arbeit, es werden jedoch Grundlagen gelegt, diese computergestützt zu beweisen, sowie die in [12] geführten Beweise zu verifizieren. Insbesondere werden die dort genutzten Techniken der Teilformelfiltration sowie des Teilformelkalküls mittels des Beweishelfers Coq formalisiert. Hierfür wurden die wichtige Lemmata, die den Zusammenhang zwischen diesen Techniken und der prinzipalen Inhabitation zeigen, implementiert. [12] nennt beide Techniken die *algorithmische Essenz* der prinzipalen Inhabitation.

1.1 Vorgehen

Zunächst wird in Kapitel 2 der einfach getypte Lambda-Kalkül als theoretische Grundlage vorgestellt. Hierfür werden zunächst λ-Terme und einfache Typen unabhängig voneinander eingeführt. Anschließend werden diese in einem Kalkül miteinander verbunden. Zusätzlich werden grundlegende Lemmata aufgestellt und bewiesen, die helfen ein fundiertes Verständnis des einfach getypten Lambda-Kalküls zu erhalten, auf das in den nachfolgenden Kapiteln aufgebaut werden kann. Die Struktur des Kapitels, sowie ein Großteil der Definitionen folgt [19].

Auf diesen Grundlagen aufbauend, wird in Kapitel 3 die prinzipale Inhabitation sowie die in [12] vorgestellte Teilformelfiltration und der Teilformelkalkül präsentiert. Beide Techniken werden hierfür zunächst eingeführt und dann jeweils ein Bezug zur prinzipalen Inhabitation hergestellt. Hierfür werden die in [12] aufgestellten Lemmata sowie Beweise vorgestellt und die Ideen zu Letzteren verdeutlicht.

In Kapitel 4 wird die im Rahmen dieser Arbeit angefertigte Formalisierung der Theorie beschrieben. Hierfür werden die in der Implementierung verwendeten Datentypen sowie Beweismechanismen vorgestellt um sowohl Lemma 14, als auch Lemma 25 aus [12] sowie der hierfür notwendigen Hilfslemmata zu verifizieren. Lemma 14 stellt einen Bezug zwischen der Teilformelfiltration und der prinzipalen Inhabitation her; Lemma 25 ist ein wichtiges Lemma, um einen Bezug zwischen dem Teilformelkalkül und der prinzipalen Inhabitation herzustellen[3]. Die Beweise der weiteren Lemmata wurden aus Zeitgründen nicht implementiert, jedoch wurden diese zum Teil ohne Beweis formalisiert. Die Beweise und Formalisierungen werden mittels des Beweishelfers Coq in der Version 8.8.0 formuliert. Hierfür wird zudem die Bibliothek Autosubst verwendet, die speziell für die Handhabung von Substitutionen in Termen entwickelt wurde. In dem Kapitel werden nicht die Implementierungen der Beweise angegeben, sondern lediglich die Formalisierung der Lemmata aus Kapitel 3. Dies entspricht an dieser Stelle einer

[3] Diese entsprechen Lemma 3.18 und Lemma 3.32 in dieser Arbeit

API-Dokumentation und soll sowohl helfen, die hier vorgestellte Implementierung nachzuvollziehen, als auch helfen, auf der Implementierung aufzubauen, um beispielsweise diese zu vervollständigen. Auch wird nicht jedes verwendete Lemma und jede verwendete Konstruktion explizit vorgestellt, da viele lediglich als Hilfestellung für die Implementierung der *wichtigen* Lemmata fungieren. Von diesen Hilfskonstruktionen werden nur die Wichtigsten vorgestellt, sodass dies dem Verständnis der Beweisführung hilft.

Abschließend wird ein Fazit gezogen. Insbesondere wird evaluiert, inwieweit sich das Verifizieren in Coq und das schriftliche Beweisen unterscheiden. Zudem wird ein Ausblick gegeben, wie Prinzipalität in anderen Bereichen, als der Inhabitation betrachtet werden kann.

2 Der einfach getypte Lambda-Kalkül

In diesem Kapitel werden die Grundlagen des einfach getypten Lambda-Kalküls ($\lambda_{\to}$) vorgestellt, die wir in den späteren Kapiteln benötigen. Der einfach getypte Lambda-Kalkül setzt sich aus den λ-Termen und den einfachen Typen zusammen. Zunächst werden die Terme eingeführt, sowie eine Vorschrift, die es erlaubt Berechnungen über λ-Terme durchzuführen. Des Weiteren werden Normalformen definiert, die insbesondere in den nachfolgenden Kapiteln wichtig werden.

Weiter werden die einfachen Typen eingeführt und diese mit den λ-Termen kombiniert, um ein Typsystem zu erhalten und den Begriff der Inhabitation zu definieren und zu untersuchen.

Zuletzt werden wir eine alternative, aber äquivalente Definition der λ-Terme kennenlernen, die es uns einfacher macht, computergestützt Aussagen über den einfach getypten Lambda-Kalkül aufzustellen und zu beweisen.

2.1 λ-Terme

λ-Terme sind eine Notation für funktionale Berechnungen. Wir werden zunächst λ-Präterme betrachten und auf Basis derer die λ-Terme definieren, die wir in diesem und dem nächsten Kapitel nutzen werden. Wir nutzen hier die Formalisierung aus [19].

Definition 2.1: λ-Präterm

Sei $\mathcal{V}$ eine abzählbar unendliche Menge an Variablensymbolen, dann sei die Menge Λ^- der λ-*Präterme* die kleinste Menge mit den folgenden Eigenschaften.

$$v \in \mathcal{V} \Rightarrow v \in \Lambda^- \qquad \text{(Variable)}$$

$$M, N \in \Lambda^- \Rightarrow (M\ N) \in \Lambda^- \qquad \text{(Applikation)}$$

$$v \in \mathcal{V} \wedge M \in \Lambda^- \Rightarrow (\lambda x.M) \in \Lambda^- \qquad \text{(Abstraktion)}$$

Notation

Applikationen binden nach links und Abstraktionen nach rechts. Wir verzichten auf Klammerung, falls Zugehörigkeit über die Bindungsregeln eindeutig ist. Des Weiteren verzichten wir auf die äußerste Klammer eines Terms und fassen mehrere Abstraktionen zu einer zusammen. Anstelle der Terme

$$(((M\ N)\ O)\ P), (\lambda x.(\lambda y.x)) \text{ und } ((\lambda x.(P\ Q))\ R)$$

schreiben wir

$$M\ N\ O\ P, \lambda x\ y.x \text{ und } (\lambda x.P\ Q)\ R$$

.

Konvention

Wir bezeichnen Terme üblicherweise mit Großbuchstaben aus der Mitte des Alphabets (z. B. M, N, P, Q), und Variablen mit Kleinbuchstaben aus den letzten Buchstaben des Alphabets (z. B. x, y, z).

Bemerkung: Abstraktionen werden häufig anonyme Funktionen genannt. Anstelle eine Funktion direkt mit einem Namen zu definieren, wie beispielsweise $f(x) = x$, können wir mithilfe von λ-Termen die Funktion unabhängig von ihrem Namen notieren. Selbstverständlich können wir den Term anschließend einen Namen zuweisen, $f = \lambda x.x$. Formal definieren wir dadurch nur die Berechnungsvorschrift einer Funktion, eine Funktion ist aber zusätzlich noch über ihren Definitions- und Wertebereich definiert. Wir werden diese Beobachtung in Abschnitt 2.2 erneut betrachten.

Beispiel 2.2

- Der Term $\lambda x.x$ entspricht der Funktion $f(x) = x$ und wird mit **I** abgekürzt.
- Der Term $\lambda x\ y.x$ entspricht der Funktion $f(x, y) = x$ und wird mit **K** abgekürzt.
- Der Term $\lambda x\ y\ z.x\ z\ (y\ z)$ entspricht der Funktion $f(x, y, z) = x(z)(y(z))$ und wird mit **S** abgekürzt.

Beispiel 2.2 (Fortsetzung)

- Der Term x entspricht der Konstanten x. Hierbei ist zu bemerkten, dass wir Konstanten als nullstellige Funktionen auffassen können.
- Der Term $y\ x$ entspricht $y(x)$ der Anwendung einer Funktion y auf einen Term x.

Bemerkung: Die Terme **S**,**K** und **I** bilden die Grundlage des SKI-Combinator-Calculus.[17]

Wenn wir den Term $\lambda x.x$ und den Term $\lambda y.y$ betrachten, so entsprechen sie den Funktionen $f(x) = x$ und $f'(y) = y$. Es ist intuitiv klar, dass sich f und f' gleich verhalten. Sie nehmen eine Eingabe und geben diese unverändert zurück. Hierbei ist es unerheblich, ob die Eingabe x oder y genannt wird. Syntaktisch sind sie jedoch verschieden. Wenn wir rein die Konstruktion über die λ-Präterme betrachten, folgt, dass $\lambda x.x \neq \lambda y.y$.

Um diese intuitive Gleichheit zu formalisieren, führen wir zunächst Ersetzungen von Variablen in Termen ein. Darauf aufbauend werden zunächst wir eine Relation einführen, die nach unserer Intuition gleiche Terme in Relation setzt, also $\lambda x.x \sim \lambda y.y$, sowie einer Definition für λ-Terme, in denen gilt $\lambda x.x = \lambda y.y$.

2.1.1 Substitution

Die Substitution ist die Grundlage des Berechnungsmodells im Lambda-Kalkül. Vereinfacht betrachtet ersetzen wir alle Variablen, die durch eine Abstraktion eingeführt werden durch einen Term, der durch die rechte Seite einer Applikation gegeben wird. Betrachten wir dazu zunächst, wie wir intuitiv einen *Rechenschritt* mittels mathematischen Funktionen durchführen. Sei $f : \mathbb{N} \to \mathbb{N}$ eine Funktion, die mit $f(x) = x + 3$ definiert ist. Wollen wir nun $f(3)$ berechnen, ersetzen wir zunächst alle Vorkommen von x durch 3 und erhalten die nullstellige Funktion $3 + 3$, welche wir später weiter zu 6 auswerten können.

Wir übertragen nun diese Intuition auf unsere λ-Präterme. Sei c_3 ein λ-Präterm, der der natürlichen Zahl 3 entspricht und A_+ ein λ-Präterm, der der (natürlichen) Addition

entspricht[1], dann können wir den folgenden Term formulieren, der der Funktion f entspricht

$$\lambda x.A_+\ x\ c_3.$$

Wir können im *inneren* λ-Term alle x durch c_3 ersetzen, um den Term $A_+\ c_3\ c_3$ zu erhalten. Wir schreiben $M[x/N]$ für den Term M, in dem die Variable x durch den Term N ersetzt wurde. Es gilt somit $(A_+\ x\ c_3)[x/c_3] = A_+\ c_3\ c_3$. Die Substitution verhält sich somit genau, wie unser intuitives Verständnis.

Aufpassen müssen wir bei geschachtelten Abstraktionen. Wir erinnern uns, dass in unserem intuitiven Verständnis gelten soll, dass $\lambda x.x \sim \lambda y.y$. Des Weiteren soll gelten, dass falls $M \sim N$ auch für alle x und P gelten soll, dass $M[x/P] \sim N[x/P]$. Betrachten wir die beiden Substitutionen $(\lambda x.x)[x/P]$ und $(\lambda y.y)[x/P]$. Wenn wir einfach alle Vorkommen von x einer Variable durch P ersetzen, erhalten wir die Terme $\lambda x.P$ und $\lambda y.y$. Klar ist, dass die Terme vor der Substitution intuitiv gleich sind, klar ist aber auch, dass $\lambda x.P$ und $\lambda y.y$ nicht zwangsweise gleich sind. Um dies zu verhindern, müssen wir Abstraktionen besonders behandeln. Hierfür benötigen wir eine Definition von gebundenen und freien Variablen.

Definition 2.3: Freie Variable

Sei $x \in \mathcal{V}$ und $M, N \in \Lambda^-$. Dann ist $FV : \Lambda^- \to \mathcal{P}(\mathcal{V})$ mit

$$\begin{aligned} FV(x) &= \{x\} \\ FV(M\ N) &= FV(M) \cup FV(N) \\ FV(\lambda x.N) &= FV(M) \setminus \{x\} \end{aligned}$$

eine Funktion, die einem Term seine freien Variablen zuweist. Falls $y \in FV(M)$, nennen wir y eine *freie Variable* im Term M.

Analog können wir die Menge der gebundenen Variablen definieren.

Definition 2.4: Gebundene Variable

Sei $x \in \mathcal{V}$ und $M, N \in \Lambda^-$. Dann ist $BV : \Lambda^- \to \mathcal{P}(\mathcal{V})$ mit

$$\begin{aligned} BV(x) &= \emptyset \\ BV(M\ N) &= BV(M) \cup BV(N) \end{aligned}$$

[1] Die entsprechenden Terme existieren. Die Terme c_n heißen Churchnumerale. Siehe hierfür [19]

Definition 2.4 (Fortsetzung)

$$BV(\lambda x.N) = BV(N) \cup \{x\}$$

eine Funktion, die einem Term seine gebundenen Variablen zuweise. Falls $y \in BV(M)$, nennen wir y eine *gebundene Variable* im Term M.

Bemerkung: Eine Variable kann gleichzeitig als freie und als gebundene Variable in einem Term vorkommen.

$$x \in BV(x\ \lambda x.x) \text{ und } x \in FV(x\ \lambda x.x)$$

Definition 2.5: Geschlossener Term

Ein Term M ist geschlossen, wenn $FV(M) = \emptyset$.

Wir könnten nun eine Substitution angeben, die genau die freien Variablen eines Term ersetzt. In unserem obigen Beispiel gelte dann folgendes.

$$(\lambda x.x)[x/N] = \lambda x.x \sim \lambda y.y = (\lambda y.y)[x/N]$$

Wir müssen zusätzlich noch darauf achten, dass eine Substitution keine bereits vorhandenen gebundenen Variablen als freie Variablen einführt. Betrachten wir dazu die Substitution $(\lambda x.y)[y/x]$. Wenn gelten soll, dass $\lambda x.y \sim \lambda z.y$, da auch die Funktionen $f(x) = y$ und $f'(z) = y$ identisch sind, gilt ebenfalls $(\lambda x.y)[y/x] \sim (\lambda z.y)[y/x]$, also $\lambda x.x \sim \lambda z.y$. Es ist auch klar, dass die Gleichheit $\lambda x.x = \lambda z.x$ nicht mit unserem intuitiven Verständnis übereinstimmt. Um zu verhindern, dass eine gebundende Variable als freie Variable in dem einzusetzenden Term vorkommt, können wir die gebundene Variable zuvor durch eine unbenutzte Variable ersetzen und damit eine Substitutionsvorschrift definieren, die unsere intuitive Gleichheit von Termen respektiert.

Definition 2.6: Frische Variable

Eine Variable x ist genau dann *frisch* in einem Term M, wenn sie weder als freie, noch als gebundene Variable vorkommt.

Proposition 2.7

Zu jedem Term M gibt es unendlich viele Variablen, die frisch sind.

Beweis: Jeder Term M ist endlich, damit ist die Menge der vorkommenden Variablen endlich. Eine Variablenmenge $\mathcal{V}$ ist abzählbar unendlich.

Definition 2.8: Substitution

Sei $M, P \in \Lambda^-$, $z \in \mathcal{V}$ frisch in M und P, $x, y \in \mathcal{V}$ mit $x \neq y$. Wir definieren die *Substitution* $M[x/P]$ durch Fallunterscheidung über M.

$$
\begin{aligned}
x[x/P] &= P \\
y[x/P] &= y \\
(M\ N)[x/P] &= M[x/P]\ N[x/P] \\
(\lambda y.N)[y/P] &= \lambda y.N \\
(\lambda x.N)[y/P] &= \begin{cases} \lambda z.N[x/z][y/P] & \text{Falls x} \in FV(P) \cap FV(N) \\ \lambda x.N[y/P] & \text{sonst} \end{cases}
\end{aligned}
$$

Bemerkung:

- In der Literatur gibt es für die Substitution verschiedene Notationen. Wir folgen hier [20], sodass wir eine Substitution $[a/b]$ als das Ersetzen von a durch b lesen. Es gibt auch die Lesart, sodass wir b durch a ersetzen.[11]
- Es gibt aufgrund Proposition 2.7 immer eine frische Variable z, die wir wählen können.
- Wir nennen eine Substitution durch eine Variable auch eine Umbenennung.

Das folgende Lemma beweist die Intuition, dass falls eine Variable nicht in der Definition einer Funktion (frei) vorkommt, eine Ersetzung den Term nicht beeinflusst.

Lemma 2.9

Für alle $x \in \mathcal{V}$, $M, N \in \Lambda$ gilt, falls $x \notin FV(M)$, dann gilt $M[x/P] = M$

Beweis: Durch Induktion über M.

Fall (IA) $M = y$**:**

$$x \notin FV(y) \Rightarrow x \neq y \Rightarrow y[x/P] = y$$

Fall $M = P\ Q$**:**

$$\begin{aligned} x \notin FV(P\ Q) &\Rightarrow x \notin FV(P) \wedge x \notin FV(Q) \\ &\overset{IH}{\Rightarrow} P[x/N] = P \wedge Q[x/N] = Q \\ &\Rightarrow (P\ Q)[x/N) = P\ Q \end{aligned}$$

Fall $M = \lambda y.P$ **und** $x \neq y$**:**

$$\begin{aligned} & x \notin FV(\lambda y.P) \wedge x \neq y \\ & \Rightarrow x \notin FV(P) \overset{IH}{\Rightarrow} P[x/N] = P \\ & \Rightarrow \lambda y.P[x/N] = \lambda y.P \\ & \Rightarrow (\lambda y.P)[y/N] = \lambda y.P \end{aligned}$$

Fall $M = \lambda x.N$**:**

$$\lambda x.N[x/P] = \lambda x.N$$

Wir können nun formulieren, dass zwei Terme in einer Art und Weise gleich sind, wenn sie sich nur in den Namen ihrer gebundenen Variablen unterscheiden.

Definition 2.10: α-Äquivalenz

Seien $x, y \in \mathcal{V}$ Variablen, $N, N', M, M' \in \Lambda^-$ λ-Präterme. Wir definieren die α-Äquivalenz $\equiv_\alpha$ als kleinste transitive, reflexive und symmetrische Relation (Äquivalenzrelation) mit den folgenden Eigenschaften.

$$\begin{aligned} \lambda x.M &\equiv_\alpha \lambda y.M[x/y] \\ M \equiv_\alpha M' &\Rightarrow \lambda x.M \equiv_\alpha \lambda x.M' \\ M \equiv_\alpha M' &\Rightarrow N\ M \equiv_\alpha N\ M' \end{aligned}$$

Definition 2.10 (Fortsetzung)

$$N \equiv_\alpha N' \Rightarrow N\ M \equiv_\alpha N'\ M.$$

Mit der α-Äquivalenz haben wir nun eine Relation für $\sim$, unter der die von uns als intuitiv gleich angesehenen Terme äquivalent sind. Dies können wir nun zu einem Begriff erweitern, der es uns erlaubt, diese äquivalenten Terme als gleich anzusehen.

Definition 2.11: λ-Terme

Sei $[M]_\alpha$ die Äquivalenzklasse des λ-Präterms M unter der Relation $\equiv_\alpha$, dann ist die Menge der λ-Terme Λ definiert als

$$\Lambda = \{[M]_\alpha \mid M \in \Lambda^-\}$$

Notation

Wir werden implizit die Äquivalenzklasse weglassen, sodass wir anstelle $[\lambda x.x]_\alpha \in \Lambda$, direkt $\lambda x.x \in \Lambda$ schreiben, es gilt somit $\lambda x.x = \lambda y.y$.

Bemerkung: In der Literatur werden λ-Terme häufig wie unsere λ-Präterme definiert und anstelle der Gleichheit zwischen Termen wird nur eine Äquivalenz unter der α-Äquivalenz angegeben. Unsere explizite Unterscheidung zwischen Λ und Λ^- erlaubt es uns echte Gleichheiten zwischen λ-Termen für unser intuitives Verständnis von Gleichheit zu verwenden. Siehe hierzu auch [19].

Das folgende Lemma erlaubt es uns, die Reihenfolge von Substitutionen zu ändern.

Lemma 2.12: Substitutionslemma

Seien $M, N, P \in \Lambda$ λ-Terme, $x, y \in \mathcal{V}$ Variablen mit $x \neq y$ und $x \notin FV(P)$, dann gelte :

$$M[x/N][y/P] = M[y/P][x/N[y/P]]$$

Beweis: Siehe [19].

2.1.2 β-Reduktion

Wir können mittels λ-Termen nun Funktionen und Funktionsanwendungen notieren, haben aber noch keine Vorschrift, nach der wir Berechnungen ausführen können. Genauer fehlt uns eine Vorschrift, die eine Applikation einer Abstraktion und einen anderen Term zusammenführt. Betrachten wir erneut mathematische Funktionen. Wenn wir eine einstellige Funktion $f(x) = M$ haben und zu einen zu f *passenden*[2] Wert N, können wir die Anwendung von f auf N, also $f(N)$ berechnen, indem wir in M alle x durch N substituieren. In den λ-Termen entspräche dies einem Übergang von $(\lambda x.M)\ N$ zu $M[x/N]$.

Definition 2.13: Redex

Sei $x \in \mathcal{V}$ eine Variable und $M, N \in \Lambda$ zwei λ-Terme. Wir nennen einen Term in der Form $(\lambda x.M)\ N$ einen Redex.

Definition 2.14: β-Reduktion

Sei $x \in \mathcal{V},\ M, N, M', N' \in \Lambda$, dann definieren wir die β-Reduktion $\to_\beta$ als kleinste Relation mit den folgenden Eigenschaften.

$$(\lambda x.M)\ N \to_\beta M[x/N]$$
$$M \to_\beta M' \Rightarrow \lambda x.M \to_\beta \lambda x.M'$$
$$M \to_\beta M' \Rightarrow MN \to_\beta M'\ N$$
$$N \to_\beta N' \Rightarrow MN \to_\beta M\ N'$$

Bemerkung: Die β-Reduktion ersetzt genau einen Redex in einem Term. Der transitiv-reflexive Abschluss der β-Reduktion $\twoheadrightarrow_\beta$, die sogenannte *multi-step β-Reduktion*, ersetzt beliebig viele Redexe – auch keinen – in einem Term.

Über die β-Reduktion können wir nun Rechnungen durchführen. Es ist sogar so, dass durch diese Relation der Lambda-Kalkül *Turingvollständig* ist [8][3]. Wir werden in Abschnitt 2.1.4 das Konzept *Reduktion als Berechnung* aufgreifen und beschreiben, wann eine solche Berechnung terminiert.

[2] Wir werden in Abschnitt 2.2 uns damit befassen, was es bedeutet, dass ein Wert *passend* für einen Term ist.

[3] Der Lambda-Kalkül ist sogar älter als das Konzept der Turingmaschine. Vor der Einführung der Turingmaschine wurde Berechenbarkeit als λ-Definierbarkeit über den Lambda-Kalkül definiert[8].

Beispiel 2.15

- $(\lambda x\ y.x)\ z \to_\beta (\lambda y.x)[x/z] = \lambda y.x[x/z] = \lambda y.z$
- $(\lambda x.x\ y)\ (\lambda x.x) \to_\beta (x\ y)[x/\lambda x.x] = (\lambda x.x)\ y \to_\beta x[x/y] = y$

2.1.3 η-Reduktion

Zwei weitere Relation, die wir einführen, sind die η-Reduktion und die η-Expansion. Die Grundidee hierbei ist es, *unnötige* Abstraktionen zu vermeiden, bzw. explizit einen Term in eine weitere Abstraktion einzubetten. Wenn wir eine Abstraktion haben, die ihre abstrahierte Variable nur an ihren inneren Term weitergibt, können wir die Abstraktion auch weglassen, ohne das Verhalten des Terms zu ändern. Andersherum können wir jeden Term auch in zusätzliche Abstraktionen einbetten.

Beispiel 2.16

Betrachten wir den Term $(\lambda x.N\ x)\ M$ mit $x \notin FV(N)$. Nach den Regeln der β-Reduktion gilt $(\lambda x.N\ x)\ M \to_\beta (N\ x)[x/M] = NM$, jedoch gilt nicht $\lambda x.Nx \to_\beta N$.

Es liegt nun nahe, dass $\lambda x.Nx$ und N aus Beispiel 2.16 in einer geeigneten Relation zueinander stehen.

Definition 2.17: η-Reduktion

Sei $x, z \in \mathcal{V}$ eine Variable und $M, N \in \Lambda$ ein λ-Term. Sei weiter $z \notin FV(N)$, dann definieren wir die *η-Reduktion* $\to_\eta$ als kleinste Relation mit den folgenden Eigenschaften.

$$\lambda z.N\ z \to_\eta N$$
$$M \to_\eta M' \Rightarrow \lambda x.M \to_\eta \lambda x.M'$$
$$M \to_\eta M' \Rightarrow MN \to_\eta M'\ N$$
$$N \to_\eta N' \Rightarrow MN \to_\eta M\ N'$$

Analog dazu lässt sich die η-Expansion definieren.

Definition 2.18: η-Expansion

Sei $x, z \in \mathcal{V}$ eine Variable und $M, N \in \Lambda$ ein λ-Term. Sei weiter $z \notin FV(N)$, dann definieren wir die *η-Expansion* $\to_\eta^{-1}$ als kleinste Relation mit den folgenden Eigenschaften.

$$\begin{aligned} N &\to_\eta^{-1} \lambda z.N\ z \\ M \to_\eta^{-1} M' &\Rightarrow \lambda x.M \to_\eta^{-1} \lambda x.M' \\ M \to_\eta^{-1} M' &\Rightarrow MN \to_\eta^{-1} M'\ N \\ N \to_\eta^{-1} N' &\Rightarrow MN \to_\eta^{-1} M\ N' \end{aligned}$$

Bemerkung: Analog zur β-Reduktion sind $\twoheadrightarrow_\eta$ und $\twoheadrightarrow_\eta^{-1}$ die transitiv-reflexive Abschlüsse der η-Reduktion und η-Expansion.

2.1.4 Normalformen

Aufbauend auf den eingeführten Reduktionsregeln können wir nun Normalformen einführen, also Formen, in den die jeweilige Reduktion nicht mehr anwendbar ist. Wenn ein Reduktionsschritt ein Berechnungsschritt ist, ist eine Normalform das Ergebnis der Berechnung.

Definition 2.19: β-Normalform

Ein Term M, zu dem es keinen Term N gibt, sodass $M \to_\beta N$, ist in β-Normalform.

Bemerkung: Es gibt λ-Terme, die keine β-Normalform haben, also deren Berechnung nicht terminiert. Dies ist insbesondere ein wichtiges Kriterium für die Turingvollständigkeit des Lambda-Kalküls. Betrachten wir den Term $\Omega = (\lambda x.x\ x)\ (\lambda x.x\ x)$. Der Term enthält einen Redex, somit lässt er sich β-reduzieren. Eine Reduktion führt jedoch direkt wieder zu dem Term Ω

$$\begin{aligned} \Omega &= (\lambda x.x\ x)\ (\lambda x.x\ x) \to_\beta (x\ x)[x/\lambda x.x\ x] \\ &= x[x/\lambda x.x\ x]\ x[x/\lambda x.x\ x] = (\lambda x.x\ x)\ (\lambda x.x\ x) \\ &= \Omega \end{aligned}$$

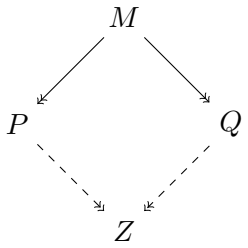


Abbildung 2.1: Diamanteigeschaft

Bemerkung: Terme, die mehrere Redexe haben, stehen mit mehreren verschiedenen λ-Termen gemäß der β-Reduktion in Relation. Die β-Reduktion ist somit nicht eindeutig.

Auch zu der η-Reduktion können wir einen Normalformbegriff definieren.

Definition 2.20: η-Normalform

Ein Term M, zu dem es keinen Term N gibt, sodass $M \to_\eta N$ ist in η-Normalform.

Wir werden in Abschnitt 2.3.2 eine Art Normalformbegriff für die η-Expansion kennenlernen.

Im Folgenden werden Wir feststellen, dass es für das Ergebnis der Berechnung eines λ-Terms unerheblich ist, in welcher Reihenfolge wir Redexe auflösen. Zunächst führen wir eine Eigenschaft ein, die besagt, dass die Wahl des Abzuleitenden Subterms in einer Reduktion in der Hinsicht nicht relevant ist, dass wir immer zu einem gemeinsamen Ergebnisterm gelangen können.

Definition 2.21: Diamanteigenschaft

Seien $M, P, Q \in \Lambda$ λ-Terme, dann sagen wir, dass eine transitiv-reflexive Relation $\twoheadrightarrow$ die Diamanteigenschaft erfüllt oder konfluent ist, wenn

$$M \twoheadrightarrow P \wedge M \twoheadrightarrow Q \Rightarrow \exists Z \in \Lambda . P \twoheadrightarrow Z \wedge Q \twoheadrightarrow Z.$$

Siehe hierfür auch Abbildung 2.1.

Proposition 2.22: Church-Rosser

$\twoheadrightarrow_\beta$ und $\twoheadrightarrow_\eta$ erfüllen die Diamanteigenschaft.

Beweis: Siehe [19].

Dies können wir nun erweitern, um festzustellen, dass eine Normalform – also ein Ergebnis einer Berechnung – nicht davon abhängt, in welcher Reihenfolge Redexe ausgewertet werden.

Lemma 2.23: Eindeutigkeit von Normalformen

Seien N_1 und N_2 zwei Normalformen eines Terms M zu einer Relation, dessen transitiv-reflexiver Abschluss die Diamanteigenschaft erfüllt, dann ist $N_1 = N_2$.

Beweis: Folgt direkt aus der Diamanteigenschaft. Ein Term in Normalform steht nur mit sich selbst in Relation, also $N_1 \twoheadrightarrow Z \Rightarrow Z = N_1$ und $N_2 \twoheadrightarrow Z \Rightarrow N_2 = Z$. Daher gilt $Z = N_1 = N_2$.

2.2 Einfache Typen

Betrachten wir erneut Funktionen. Funktionen sind üblicherweise nicht nur durch ihren Funktionsrumpf definiert, sondern auch durch ihren Definitionsbereich (Domain) und Wertebereich (Codomain). Eine vollständige Definition für eine Funktion wäre die folgende:

$$f : \mathbb{N} \to \mathbb{N}$$
$$f(x) = x + 3$$

Wir können hier $\mathbb{N} \to \mathbb{N}$ als Typ für die Funktion f sehen. Die Menge der natürlichen Zahlen ist dann ein Typ für die Eingabe x sowie für alle Ergebnisse von f. Betrachten wir den Aufbau des Typen $\mathbb{N} \to \mathbb{N}$. Dieser besteht aus zwei Komponenten, ein zweistelliger Operator $\to$ und die atomaren Typen $\mathbb{N}$.

Wir können ebenfalls der Identitätsfunktion $id(x) = x$ einen Typen geben. Die Domain und Codomain von id ist durch die Definition nicht gegeben, es ist jedoch klar, dass die Codomain zumindest die Domain enthalten muss. Sei die Domain vom id eine beliebige Menge A, so ist die Codomain mindestens eine Menge B mit $A \subseteq B$. Insbesondere gilt $id : A \to A$. Dieses Prinzip können wir auf λ-Terme erweitern. Es gibt

es viele Typsysteme, die den Lambda-Kalkül typen, wir betrachten hier die einfachen Typen.

Definition 2.24: Einfache Typen

Sei $\mathbb{A}$ eine abzählbar unendliche Menge an Typatomen, dann ist die Menge $\mathbb{T}$ der einfachen Typen definiert als kleinste Menge mit den folgenden Eigenschaften:

$$a \in \mathbb{A} \Rightarrow a \in \mathbb{T}$$
$$\alpha, \beta \in \mathbb{T} \Rightarrow \alpha \to \beta \in \mathbb{T}$$

Konvention

Wir bezeichnen Typen mit kleinen griechischen Buchstaben (z. B. α, β aber auch σ, τ, ρ), Typatome mit kleinen Buchstaben aus den ersten Buchstaben des Alphabets (z. B. a, b, c)

Definition 2.25: Teiltyp

Sei $\rho \in \mathbb{T}$ ein einfacher Typ, dann ist die Menge $\mathrm{codom}(P_\rho)$ seiner Teiltypen folgendermaßen induktiv definiert.

$$\begin{aligned} \mathrm{codom}(P_b) &= b && \text{für } b \in \mathbb{A} \\ \mathrm{codom}(P_{\sigma\to\tau}) &= \{\sigma \to \tau\} \cup \\ &\quad \mathrm{codom}(\sigma) \cup \\ &\quad \mathrm{codom}(\tau) \end{aligned}$$

Falls $\rho' \in \mathrm{codom}(\rho)$, aber $\rho \neq \rho'$, dann nennen wir ρ' einen echten Teiltypen von ρ.

Bemerkung:

- Die Benennung der Menge der Teiltypen deutet bereits an, dass P_ρ eine Funktion darstellt. Dies ist der Fall und sie wird in Definition 3.19 definiert.
- In der Literatur wird anstelle von Teiltypen auch von Subformelen gesprochen. Dies hängt mit dem *Curry-Howard-Isomorphismus* zusammen, der besagt, dass Typen Formeln darstellen. Wir werden dies in Abschnitt 2.3.1 betrachten.

Analog zu den Variablen in λ-Termen können wir für Typatome in einfachen Typen eine Definition für unbenutzte, frische Variablen aufstellen.

Definition 2.26: Frisches Typatom

Wir nennen ein Atom a *frisch* in einem Typen ρ, wenn es nicht als Teiltyp von ρ auftaucht.

2.3 Typisierung

Wir benötigen nun einen Kalkül, mit dem wir Terme und Typen in Relation setzen können. Es ist auffällig, dass $\lambda x.M$ eine (mindestens einstellige) Funktion darstellt und $\alpha \to \beta$ ein Funktionstyp ist. Interpretieren wir α als Domain und β als Codomain der Funktion, müssen alle Terme N, auf die die Funktion angewendet wird, den Typen α haben und der Term $M[x/N]$ den Typen β. Da N den Typen α hat, muss x in M auch den Typen α haben. Wir müssen uns also merken, dass in M die Variable x mit α getypt wird. Dazu müssen wir uns einen Typkontext merken, der Variablen Typen zuweist. Wir führen hierfür eine Relation $\Gamma \vdash M : \rho$ ein, die aussagt, dass zu einem Typkontext Γ der Term M mit dem Typen ρ getypt werden kann.

Definition 2.27: Typkontext

Ein Typkontext Γ ist eine Relation mit $\Gamma \subseteq \mathcal{V} \times \mathbb{T}$, bei der alle Elemente aus $\mathcal{V}$ maximal einmal in der Domain der Relation vorkommen dürfen.

Notation

Wir schreiben Elemente (x, ρ) aus Γ als $x : \rho$. Des Weiteren schreiben wir $\Gamma, x : \rho$ anstelle von $\Gamma \cup (x, \rho)$. Hierbei ist zu beachten, dass die Bedingung an Typkontexte nicht verletzt wird, da durch das Hinzufügen eine Variable mit mehreren Typen getypt werden könnte.

Definition 2.28: Typisierung

Seien $\Gamma \subseteq \mathcal{V} \times \mathbb{T}$ ein Typkontext, $\sigma, \tau, \rho \in \mathbb{T}$ einfache Typen, $M, N \in \Lambda$ λ-Terme und $x \in \mathcal{V}$ eine Variable. Wir definieren die Typisierung $\_ \vdash \_ : \_$ anhand folgender Inferenzregeln.

$$\frac{}{\Gamma, x : \rho \vdash x : \rho} \text{ (var)}$$

Definition 2.28 (Fortsetzung)

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau} \text{ (abs)}$$

$$\frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash M\ N : \tau} \text{ (app)}$$

Wir bezeichnen einen Baum $\mathcal{D}$ mit Wurzel $\Gamma \vdash M : \rho$, der sich durch die Anwendung der obigen Inferenzregeln ergibt, als Typableitung. Wenn $\mathcal{D}$ die Typableitung zu der Typisierung $\Gamma \vdash M : \rho$ ist, schreiben wir auch $\mathcal{D} \rhd \Gamma \vdash M : \rho$.

Notation

Anstelle von $\emptyset \vdash M : \rho$ schreiben wir auch $\vdash M : \rho$.

Jede Art, einen λ-Term M zu konstruieren, korrespondiert zu genau einer Typisierungsregel. Betrachten wir den äußersten Konstruktor von M, lässt sich ablesen, welche Regel genutzt wurde, um M zu typen. Dies wird im folgenden Generierungslemma formalisiert.

Lemma 2.29: Generierungslemma

Sei $\Gamma \subseteq \mathcal{V} \times \mathbb{T}$ ein Typkontext, $M \in \Lambda$ ein λ-Term und $\rho \in \mathbb{T}$ ein einfacher Typ. Gelte weiter $\Gamma \vdash M : \rho$. Dann gilt

- falls $M = x$, $x : \rho \in \Gamma$
- falls $M = \lambda x.N$, $\rho = \sigma \to \tau$ sodass $\Gamma, x : \sigma \vdash N : \tau$
- falls $M = P\ Q$, es existiert ein σ, sodass $\Gamma \vdash M : \sigma \to \rho$ und $\Gamma \vdash N : \sigma$

Beweis: In jedem Beweisschritt ist es eindeutig, welche Regel gewählt wird.

Definition 2.30: Typbar

Terme, zu denen es ein Γ und ein ρ gibt, sodass $\Gamma \vdash M : \rho$, nennen wir typbar.

Beispiel 2.31

1. Der Term $\mathbf{I}$ ist typbar mit $\vdash \mathbf{I} : \alpha \to \alpha$
2. Der Term $x\ x$ ist nicht typbar.

Beweis:

1. $$\dfrac{\dfrac{}{x : \alpha \vdash x : \alpha}\ (var)}{\vdash \lambda x.x : \alpha \to \alpha}\ (abs)$$
2. Es ist klar, dass $x\ x$ in einem Kontext, der nicht x enthält, nicht typbar ist. Nehmen wir an, dass es eine Typisierung $\Gamma, x : \alpha \vdash x\ x : \rho$ gibt. Laut Lemma 2.29 gibt es ein σ, sodass $\Gamma, x : \alpha \vdash x : \sigma \to \rho$ und $\Gamma, x : \alpha \vdash x : \sigma$.

   Weiter folgt nach Lemma 2.29, dass $\alpha = \sigma = \sigma \to \rho$ gelten muss. Die Gleichung $\sigma = \sigma \to \rho$ kann für endliche Typen nicht erfüllt werden, und Definition 2.24 erlaubt nur endliche Typen.

Lemma 2.32: Subtypbar

Sei $M \in \Lambda$ ein λ-Term, der typbar ist. Dann ist jeder Subterm von M ebenfalls typbar.

Beweis: Folgt direkt aus Lemma 2.29.

Definition 2.33: Inhabitanten

Falls $\vdash M : \rho$, nennen wir M einen Inhabitanten von ρ und ρ inhabitiert.

Lemma 2.34: Subject-Reduction

Falls $\Gamma \vdash M : \rho$ und $M \twoheadrightarrow_\beta N$, dann $\Gamma \vdash N : \rho$.

Beweis: Siehe [19].

Bemerkung: Die umgekehrte Richtung gilt nicht. Betrachten wir die Terme $\mathbf{I}$ und $\mathbf{K}\ \mathbf{I}\ \Omega$. Es gilt $\mathbf{K}\ \mathbf{I}\ \Omega \to_\beta \mathbf{I}$ und $\vdash \mathbf{I} : \alpha \to \alpha$, es gilt jedoch nicht, dass $\vdash \mathbf{K}\ \mathbf{I}\ \Omega : \alpha \to \alpha$. Es ist sogar so, dass $\mathbf{K}\ \mathbf{I}\ \Omega$ nicht typbar ist, da $x\ x$ ein Subterm von Ω ist und $x\ x$ nicht typbar ist. Siehe Lemma 2.32 und Beispiel 2.31.

Definition 2.35: Typsubstitution

Eine Funktion $Su : \mathbb{A} \to \mathbb{T}$ nennen wir Typsubstitution. Wir können eine Substitution Su zu einer Funktion Su^* homomorph auf Typen fortsetzen.

$$Su^* : \mathbb{T} \to \mathbb{T}$$

$$Su^*(\rho) = \begin{cases} Su(\rho) & \text{falls } \rho \in \mathbb{A} \\ Su^*(\sigma) \to Su^*(\tau) & \text{falls } \rho = \sigma \to \tau \end{cases}$$

Weiter können wir eine auf Typen fortgesetzte Substitution Su^* auf Typkontexte zu einer Funktion Su^{**} fortsetzen.

$$Su^{**} : (\mathbb{T} \times \mathcal{V}) \to (\mathbb{T} \times \mathcal{V})$$
$$Su^{**}(\Gamma) = \{x : Su^*(\rho) \mid x : \rho \in \Gamma\}$$

Notation

Da Su^{**}, Su^* und Su in ihren Domain disjunkt, bzw. im Fall von Su und Su^* über einander definiert sind, schreiben wir für alle nur Su.

Wir können noch einen Zusammenhang zwischen geschlossenen Termen und Termen, die sich in einem leeren Kontext typen lassen feststellen.

Lemma 2.36

Sei M ein λ-Term und ρ ein einfacher Typ. Falls $\vdash M : \rho$, dann ist M ein geschlossener Term.

Beweis: Wir zeigen die allgemeinere Aussage, dass ein Typkontext mindestens die freien Variablen in M typen muss, durch Induktion über M. Durch Lemma 2.29 können wir direkt auf die Regel schließen, die für jeden Term genutzt wurde.

Lemma 2.36 (Fortsetzung)

Induktionsanfang, $\Gamma \vdash x : \rho$**.** Nach (var) muss $x : \rho \in \Gamma$ sein.

Induktionsschritt, $\Gamma \vdash \lambda x.N : \sigma \to \tau$**.** Unsere Induktionshypothese gilt für $\Gamma, x : \sigma \vdash N : \tau$. Im Term $\lambda x.N$ ist die Variable x nicht frei. Über das Generierungslemma (Lemma 2.29) wissen wir, dass ein Kontext, der N typt, x mit σ typt und ein Kontext, der $\lambda x.N$ typt, derselbe Kontext ohne Eintrag für x ist. Somit fällt genau die freie Variable aus dem Kontext weg, die durch die Abstraktion gebunden wird.

Induktionsschritt, $\Gamma \vdash N\ M : \tau$**.** Laut Induktionshypothese gilt typt Γ genau die freien Variablen von M und die von N. Da $FV(M\ N) = FV(M) \cup FV(N)$ typt Γ auch die freien Variablen von $M\ N$.

2.3.1 Intuitionistisch Propositionale Logik

Es ist wohlbekannt, dass Typsysteme und Logik eng miteinander verzahnt sind. Betrachten wir einen Typen $\sigma \to \tau$ und gehen wir davon aus, dass es einen Term M und eine Umgebung Γ gibt, sodass $\Gamma \vdash M : \sigma \to \tau$ gilt. Intuitiv nimmt M nun einen Term von Typen σ und konstruiert einen Term von Typen τ. Da wir über unsere Annahme wissen, dass ein M existiert, wissen wir, dass, wenn ein Term x vom Typen σ existiert, dass auch ein Term vom Typen τ existiert. Dies können wir als logische Implikation lesen: falls σ inhabitiert wird, so auch τ. Dieses intuitive Verständnis wird im Curry-Howard-Isomorphismus formalisiert [19]. Dieser sagt vereinfacht, dass wir Typen als logische Aussagen und Terme als Beweise für ihre Typen sehen können.

Wir betrachten hier die Intuitionistische Propositionale Logik (IPC[4]). Die Grundidee der IPC ist, dass jede Aussage konstruktiv bewiesen werden muss. Ein häufiges Beispiel um aufzuzeigen, wie ein Beweis nicht konstruktiv ist, ist ein Beweis für die folgende Aussage.

Lemma 2.37

Es gibt zwei irrationale Zahlen x und y, sodass x^y rational ist.

[4]Englisch: *Intuitionistic propositional calculus*

Lemma 2.37 (Fortsetzung)

Beweis: Betrachten wir $x = y = \sqrt{2}$. Es ist klar, dass x und y irrational sind. Falls $\sqrt{2}^{\sqrt{2}} \in \mathbb{Q}$, ist die Aussage bewiesen.
Ansonsten gilt $\sqrt{2}^{\sqrt{2}} \notin \mathbb{Q}$. Wir wählen also $x = \sqrt{2}^{\sqrt{2}}$ und $y = \sqrt{2}$. Es gilt $\left(\sqrt{2}^{\sqrt{2}}\right)^{\sqrt{2}} = \sqrt{2}^{\sqrt{2}\cdot\sqrt{2}} = \sqrt{2}^2 = 2 \in \mathbb{Q}$.

Wir wissen somit, dass Lemma 2.37 gilt, jedoch haben wir die Werte für x und y nicht konstruiert. Einen solchen Beweis nennen wir nicht konstruktiv. Im Kern erlaubt uns das Axiom $P \vee \neg P$ nicht konstruktive Beweise zu formulieren. Die IPC nutzt dieses Axiom nicht. Um $P \vee \neg P$ zu beweisen, müssen wir eine der beiden Teile der Disjunktion beweisen.

Syntaktisch gleicht die IPC der klassischen Aussagenlogik. Wir haben Konjunktionen ($\wedge$), Disjunktionen ($\vee$), Negationen ($\neg$), Implikationen ($\to$), sowie die Konstanten für die Wahrheit ($\top$) und die Falschheit ($\bot$) über einer Variablenmenge $\mathcal{V}$. An dieser Stelle betrachten wir jedoch nur das implikative Fragment der IPC ($\mathrm{IPC}_{\to}$), also die Beschränkung der IPC auf Implikation und Variablen. Durch Betrachten der Ableitungsregeln für $\mathrm{IPC}_{\to}$ können wir sehen, wie dieser mit den Typisierungsregeln des einfach getypten Lambda-Kalküls zusammenhängt

Definition 2.38: Ableitungsregeln für $\mathrm{IPC}_{\to}$

Sei $\mathcal{V}$ eine Variablenmenge, $\Delta \subseteq \mathcal{V}$ und $\varphi, \psi \in \mathrm{IPC}_{\to}$ intuitionistische Formeln, dann definieren wir die Ableitungsregel $\_ \vdash \_$ für das implikative Fragment der intuitionistisch propositionalen Logik anhand der folgenden Inferenzregeln:

$$\frac{}{\Delta, \varphi \vdash \varphi}\ (\text{hyp})$$

$$\frac{\Delta, \varphi \vdash \psi}{\Delta \vdash \varphi \to \psi}\ (\text{DT})$$

$$\frac{\Delta \vdash \psi \to \varphi \quad \Delta \vdash \psi}{\Delta \vdash \varphi}\ (\text{MP})$$

Wir können sehen, dass jede Ableitungsregel aus Definition 2.28 einer Ableitungsregel aus Definition 2.38 entspricht. Wir können sogar noch weitergehen und eine Äquivalenz zwischen $\mathrm{IPC}_{\to}$ und dem einfach getypten Lambda-Kalkül feststellen.

Theorem 2.39: Curry-Howard Isomorphismus[19]

1. Falls $\Gamma \vdash M : \varphi$ in $\lambda_{\to}$, dann $\{\psi \mid x : \psi \in \Gamma\} \vdash \varphi$ in $\mathrm{IPC}_{\to}$.
2. Falls $\Delta \vdash \varphi$ in $\mathrm{IPC}_{\to}$, dann gibt es ein Γ mit $\{\psi \mid x : \psi \in \Gamma\} = \Delta$ und ein M, sodass $\Gamma \vdash M : \varphi$ in $\lambda_{\to}$.

Beweis: Siehe [19].

Wir können somit Resultate des einfach getypten Lambda-Kalküls direkt auf Resultate in der Logik übertragen.

2.3.2 η-lange Normalform

Mittels der einfachen Typen können wir auch eine Art Normalform für die η-Expansion finden, die η-lange Normalform. Intuitiv betrachtet ist ein Term M in η-langer Normalform, wenn alle Subterme von M, die Funktionen darstellen, alle Parameter erhalten. Ein Problem hierbei ist es, zu identifizieren, wie viele Parameter ein Term erhalten kann. Naiv könnten wir in jedem Subterm von M für jede (freie) Variable x die maximale Anzahl Parameter zählen, auf die x angewendet wird. Es kann jedoch sein, dass durch β-Reduktion ein Term in einen Kontext eingebettet wird, in dem er auf weitere Terme angewendet wird. Wir müssen somit auch Redexe beachten und möglicherweise auflösen. Auf diese Weise ist eine Normalform nicht einfach zu finden. Eine Lösung ist hierfür, uns nur auf Terme einzuschränken, die bereits in β-Normalform sind.

Weiter können wir zwar anhand eines Typen ablesen, wie viele Eingaben ein Term verarbeitet, ein Typ für einen Term ist jedoch nicht eindeutig. Betrachten wir hierzu den Term $\mathbf{I}$ mit dem Typen $a \to a$ für $a \in \mathbb{A}$. Unter diesem Typen hat der Term genau eine Eingabe. Typen wir jedoch denselben Term mit dem Typen $(a \to b) \to a \to b$ für $a, b \in \mathbb{A}$, so sehen wir, dass $\mathbf{I}$ unter diesem Typen zwei Eingaben erwartet. Es hängt somit von dem Typ eines Terms ab, ob dieser maximal appliziert ist.

Grundsätzlich können wir feststellen, dass ein Term M mit $\vdash M : \sigma_1 \to \ldots \sigma_n \to \tau$ mindestens n Eingaben erwartet.

Bemerkung: Wir sagen *mindestens*, da τ ein Pfeiltyp $\sigma_{n+1} \to \tau'$ sein kann. Dadurch hätte die Funktion mindestens $n+1$ Parameter. Falls τ ein Typatom ist, wissen wir, dass der Term M genau n Parameter akzeptiert.

Wir werden nun eine Typrelation einführen, die genau die Terme in β-Normalform typen, in denen alle Subterme zu ihrem entsprechenden Typen maximal expandiert sind.

Definition 2.40: Lang typbar[12]

Wir nennen einen Term *lang* zu einem Typen, wenn er mit nur mit der Regel (abs) aus Definition 2.28 und der folgenden Regel in einem leeren Kontext typbar ist.

$$\frac{\Gamma, x : \sigma_1 \to \ldots \to \sigma_n \to a \vdash M_i : \sigma_i \text{ für } i = 1 \ldots n}{\Gamma, x : \sigma_1 \to \ldots \to \sigma_n \to a \vdash x\ M_1\ \ldots\ M_n : a}\ \text{(long)}$$

Die Menge $\text{Long}(\rho)$ enthält alle Terme, die lang zu ρ sind. Wir nennen einen Term M in $\text{Long}(\rho)$ einen Term in η-langer Normalform.

Bemerkung:

- $\text{Long}(\rho)$ enthält nur Terme in β-Normalform, da wir nicht erlauben, Redexe zu typen. Ein typbarer Term ist entweder eine Abstraktion oder eine Liste an Applikationen, in der der erste Term eine Variable ist.
- Es ist irrelevant, zu welchem Typen ein Term lang ist, um in η-langer Normalform zu sein. Solange ein Typ ρ existiert, sodass $M \in \text{Long}(\rho)$, ist M in η-langer Normalform.
- Der Begriff η-lange Normalform ist irreführend, da es sich weder um eine Normalform für die η-Reduktion, noch um eine Normalform für die η-Expansion handelt. Genauer existiert keine Normalform für die η-Expansion, da der Expansionsschritt beliebig häufig wiederholt werden kann. Korrekt müsste man von einem Term in η-langer β-Normalform sprechen.
- Es scheint zwar so, dass wir keine Regel haben, die eine leeren Hypothese hat, betrachten wir aber die Regel (long) mit $n = 0$, fällt die Hypothese vollständig weg. Die resultierende Regel gleicht in ihrer Form der (var)-Regel, bis darauf, dass sie Variablen nur atomaren Typen zuweist.

Dass wir die Regeln (app) und (var) durch (long) ersetzen, führt uns zu einer Einschränkung der Typisierung. Es erlaubt keine neuen Typisierungen, wie das folgende Lemma zeigt.

Lemma 2.41

Sei $M \in \Lambda$ ein λ-Term, $\rho \in \mathbb{T}$ ein einfacher Typ und gilt $M \in \text{Long}(\rho)$, dann gilt auch $\vdash M : \rho$.

Beweisidee: Da sich lange Typisierung die (abs)-Regel mit der *normalen* Typisierung teilt, müssen wir nur betrachten, dass sich eine Anwendung der (long)-Regel mit den (app)- und (var)-Regeln darstellen lässt. Wir führen hier eine Induktion über n.

Induktionsanfang, $n = 0$**:** Die Hypothese ist leer, wir haben eine Regel in der Form von (var).

Induktionsschritt, $n \rightsquigarrow n + 1$**:** Wenn wir die (long)-Regel für einen Term $M = x\ M_1\ \ldots\ M_n$ mit (app) und (var) darstellen können, dann können wir auch die Applikation des Terms M auf den Term M_{n+1} durchführen. Hierfür reicht es die (app)-Regel anzuwenden.

Lemma 2.42

Seien $Su : \mathbb{T} \to \mathbb{T}$ eine Typsubstitution, $\rho, \rho' \in \mathbb{T}$ einfache Typen und $M \in \Lambda$ ein λ-Term. Falls $M \in \text{Long}(Su(\rho))$ und $\vdash M : \rho$, dann ist auch $M \in \text{Long}(\rho)$.

Beweis: Der Beweis wird per Induktion über die Länge des Terms M geführt und wird in Kapitel 4 genauer behandelt.

Lemma 2.43: Eindeutigkeit der η-langen Normalformen [14, 8A8]

Falls M ein Inhabitant von ρ in β-Normalform ist, dann gibt es einen eindeutigen Term $M^+ \in \text{Long}(\rho)$, sodass $M \twoheadrightarrow_\eta^{-1} M^+$.

Beweis: Siehe [14, 8A8].

2.3.3 Inhabitation

Vier wichtige Fragestellungen, die Typrelation $\_ \vdash \_ : \_$ induziert sind die folgenden:

1. Gegeben Γ, M und ρ, gilt $\Gamma \vdash M : \rho$? (Type checking)

2. Gegeben Γ und M, für welche Typen ρ gilt $\Gamma \vdash M : \rho$? (Typinferenz)

3. Gegeben M, gibt es einen Typen ρ und Γ, sodass $\Gamma \vdash M : \rho$? (Typbarkeit)

4. Gegeben Γ und ρ, gibt es ein M, sodass $\Gamma \vdash M : \rho$? (Typinhabitation)

Wir werden uns hier mit der Beantwortung der vierten Frage beschäftigen. Über Theorem 2.39 gleicht dies der Frage, ob eine gegebene Formel der $\mathrm{IPC}_{\to}$ unter einem gegebenen Modell zu wahr evaluiert. Wir werden zunächst eine obere Schranke der Komplexität für die Inhabitation zeigen, indem wir einen Algorithmus angeben, der in polynomiellem Platz entscheidet, ob ein gegebener Typ einen Inhabitanten hat. Weiter werden wir zeigen, dass jedes Problem in PSPACE auf die Inhabitation in $\lambda_{\to}$ reduzierbar ist. Hierfür werden wir eine Reduktion von QBF angeben.

Wir wissen über Lemma 2.43, dass, wenn ein Typ inhabitiert wird, dann gibt es auch einen Inhabitanten in η-langer Normalform. Wir werden uns darauf beschränken Inhabitanten in η-langer Normalform zu finden. Dazu betrachten wir zunächst den alternierenden Algorithmus 1. Der Operator $\boxplus$ fügt ein Tupel nur dann zu Γ hinzu, wenn keine Variable mit dem entsprechenden Typen in Γ existiert. Wir beweisen zunächst die Korrektheit und die Vollständigkeit des Algorithmus, und geben eine Beweisidee für die Platzschranke an. Details zu Letzterem finden sich in [21].

Theorem 2.44

Falls $\mathrm{INH}_{\to}(\Gamma, \rho) = M$, dann gilt $\Gamma \vdash M : \rho$.

Beweisidee: Wir betrachten hier, wie ρ aufgebaut sein kann.

Fall $\rho = a \in \mathbb{A}$ Der Algorithmus sucht in Γ ein x, das a als Zieltypen hat. Für jeden Eingabetypen von x sucht er einen Inhabitanten, auf den er x anwenden kann. Über die (long)-Regel erhalten wir einen korrekt getypten Typen.

Fall $\rho = \sigma \to \tau$ Wenn wir einen Funktionstypen inhabitieren wollen, wissen wir, dass der resultierende (lange) Term eine Abstraktion ist. Die (abs)-Regel erlaubt uns somit, den Kontext um die abstrahierte Variable zu erweitern und rekursiv einen Inhabitanten als inneren Term zu suchen.

Algorithmus 1 : $\mathrm{INH}_{\to}$

Data : Typkontext Γ, Typ ρ

Result : M, sodass $\Gamma \vdash M : \rho$, oder Fehlschlag

```
1  switch ρ do
2      case ρ = a ∈ 𝔸 do
3          Wähle x : σ_1 → ... → σ_n → a ∈ Γ
4          if n = 0 then
5              return x
6          else
7              forall i ∈ [1, ..., n] do
8                  M_i ← INH_→(Γ, σ_n)
9              return x M_1 ... M_n
10     case ρ = σ → τ do
11         Wähle y, sodass y ∉ dom(Γ)
12         P ← INH_→(Γ ⊞ (y : σ), τ)
13         return λy.P
```

Theorem 2.45

Falls $\mathrm{INH}_{\to}(\Gamma, \rho)$ fehlschlägt, gibt es keinen Term M, sodass $\Gamma \vdash M : \rho$.

Beweis: Betrachten wir, wann der Algorithmus fehlschlagen kann. Es kann sein, dass in Zeile 3 keine Variable in Γ gefunden wurde, deren Zieltyp ρ ist. Da wir wissen, dass ρ ein Typatom ist und die resultierenden Terme maximal appliziert sind, muss ein Term für ρ eine Applikation sein. Dies erfordert durch die (long)-Regel aber, dass eine Variable mit Zieltyp ρ im Kontext vorhanden ist. Dies wäre ein Widerspruch, ρ kann nicht somit inhabitiert sein.

Wir werden für das Enthaltensein und Schwere nur eine Beweisidee angeben; ein vollständiger Beweis würde an dieser Stelle den Rahmen dieser Arbeit übersteigen.

Theorem 2.46

Inhabitation in $\lambda_{\to}$ ist in PSPACE.

Beweisidee: Falls Algorithmus 1 terminiert, entscheidet er die Inhabitation für $\lambda_{\to}$ korrekt. Da wir wissen, dass APTIME = PSPACE gilt[7], genügt es zu zeigen, dass $\text{INH}_{\to}$ nach polynomiell vielen (alternierenden) Aufrufen terminiert.
Wir können feststellen, dass alle Typen in einem Aufruf an $\text{IHN}_{\to}$ entweder bereits im Kontext oder im Typen des vorhergehenden Rekursionsschritts als Teiltyp vorhanden sein muss. Es lässt sich zeigen, dass jeder Typ der Länge m maximal m Teiltypen haben kann. Somit existieren nur $m \times m$ verschiedene rekursive Aufrufe an $\text{INH}_{\to}$ für einen initialen Aufruf an $\text{INH}_{\to}(\Gamma, \rho)$, wobei m die Länge der Typen in Γ plus die Länge von ρ, also die Länge der Eingabe ist. Damit können wir die Anzahl der Aufrufe polynomiell nach oben abschätzen und die Inhabitation in $\lambda_{\to}$ ist in PSPACE enthalten.

Theorem 2.47

Inhabitation in $\lambda_{\to}$ ist PSPACE-schwer.

Beweisidee: Wir reduzieren von QBF, einem wohlbekannten Problem, das PSPACE-vollständig ist. Wir können eine Formel in QBF φ in einen Typkontext Γ_φ und einen Typen α_φ überführen, indem wir zunächst allen Teilformeln ψ aus φ einen Typen α_ψ zuordnen, sowie für jede Variable p einen Typen α_p und $\alpha_{\neg p}$.
Den Typkontext Γ_φ erstellen wir folgendermaßen.

- Falls $\varphi = p$ oder $\varphi = \neg p$, dann $\Gamma_\varphi = \emptyset$.
- Falls $\varphi = \chi \wedge \psi$, dann $\Gamma_\varphi = \Gamma_\chi \cup \Gamma_\psi \cup \{x_\varphi : \alpha_\chi \to \alpha_\psi \to \alpha_{\chi\wedge\psi}\}$
- Falls $\varphi = \chi \vee \psi$, dann $\Gamma_\varphi = \Gamma_\chi \cup \Gamma_\psi \cup \{x^l_\varphi : \alpha_\chi \to \alpha_{\chi\vee\psi}, x^r_\varphi : \alpha_\psi \to \alpha_{\chi\vee\psi}\}$
- Falls $\varphi = \forall p.\psi$, dann $\Gamma_\varphi = \Gamma_\psi \cup \{x_\varphi : (\alpha_p \to \alpha_\psi) \to (\alpha_{\neg p} \to \alpha_\psi) \to \alpha_{\forall p.\psi}\}$
- Falls $\varphi = \exists p.\psi$, dann $\Gamma_\varphi = \Gamma_\psi \cup \{x^0_\varphi : (\alpha_p \to \alpha_\psi) \to \alpha_{\exists p.\psi}, x^1_\varphi : (\alpha_{\neg p} \to \alpha_\psi) \to \alpha_{\exists p.\varphi}\}$

Falls α_φ in dem Kontext Γ_φ inhabitiert wird, so ist φ eine Tautologie[21].

Theorem 2.48

Inhabitation in $\lambda_{\rightarrow}$ ist PSPACE-vollständig.

Beweis: Über Theorem 2.47 und Theorem 2.46.

2.4 De-Bruijn-Notation

Während unsere Definition der λ-Terme gut geeignet ist, anschaulich Funktionen zu formalisieren, verkompliziert sie es, automatisiert oder computergestützt Aussagen über den Lambda-Kalkül zu formalisieren und zu beweisen. Große Probleme resultieren aus der Alphaäquivalenz als Gleichheit. Betrachten wir hier unsere Definition der Substitution. Wenn in einer Abstraktion über einer Variable durch einen Term substituiert wird, in dem diese Variable frei vorkommt, muss zunächst mit einer neuen Variable substituiert werden. Während dies im anschaulichen Fall durch die α-Äquivalenz irrelevant wird, muss im formalen computergestützten Fall diese zusätzliche Substitution immer mit berücksichtigt werden. Auch die Einschränkung auf α-äquivalente λ-Terme benötigt zusätzliche Arbeit. Wir werden hier eine Formalisierung vorstellen, unter der zwei α-äquivalente λ-Präterme gleich sind, ohne explizite die Äquivalenzklassen konstruieren zu müssen.

Wir stellen zunächst einige Vorüberlegungen an. Grundsätzlich ist es für Berechnungen im Lambda-Kalkül irrelevant, welche Variablenmenge ihm zugrunde liegt, solange die Variablenmenge abzählbar unendlich ist. Wir wählen hierfür fest die Menge der natürlichen Zahlen $\mathbb{N}$. Des Weiteren nutzen wir gebundene Variablen im Lambda-Kalkül, um eine Abstraktion zu referenzieren, über die die Variable gebunden ist, damit wir sie substituieren können, wenn die entsprechende Abstraktion in der β-Reduktion als Redex reduziert wird. Uns steht es also frei, die entsprechenden Abstraktionen auf eine andere Art zu referenzieren, als das Wiederholen der Variable in der Abstraktion. Syntaktisch können wir folgende Menge an Termen definieren.

Definition 2.49: De-Bruijn-Notation[15]

Sei $x \in \mathbb{N}$, dann sei die Menge Λ_B der λ-Terme in *de-Bruijn-Notation* die kleinste Menge mit den folgenden Eigenschaften.

$$x \in \Lambda_B$$

$$P \in \Lambda_B \land Q \in \Lambda_B \Rightarrow P\ Q \in \Lambda_B$$

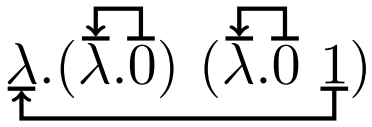


Abbildung 2.2: Grafische Darstellung der *De-Bruijn-Notation*

Definition 2.49 (Fortsetzung)

$$P \in \Lambda_B \Rightarrow \lambda.P \in \Lambda_B$$

Wir benötigen nun eine Vorschrift, mit der wir Variablen und Abstraktionen in Relation setzen können. Hierfür nutzen wir aus, dass wir Zahlen als Variablen einsetzen. Eine Variable n referenziert nun die Abstraktion, die von ihr n Abstraktionen entfernt ist. Siehe hierfür Abbildung 2.2 und Beispiel 2.50.

Beispiel 2.50

- $\mathbf{I} = \lambda.0$
- $\mathbf{K} = \lambda.\lambda.0$
- $\mathbf{S} = \lambda.\lambda.\lambda.(2\ 0)\ (1\ 0)$

In De-Bruijn-Notation sind verschiedene λ-Terme auch für unsere intuitive Gleichheit verschieden, wir benötigen somit keine α-Äquivalenz. Substitution, β-Reduktion, η-Reduktion, η-Expansion, sowie die Typisierung lassen sich analog zu den üblichen λ-Termen definieren. Wir werden in Kapitel 4 diese Relationen explizit definieren.

3 Prinzipalität

In diesem Kapitel werden zunächst die Prinzipalität und die prinzipale Inhabitation eingeführt, um darauf aufbauend zwei Methoden vorzustellen, die helfen, automatisiert zu entscheiden, ob ein gegebener Typ prinzipal zu einem gegebenen Term ist.

In Abschnitt 3.2 und 3.3 wird an vielen Stellen auf ausführliche Beweise verzichtet, da diese in der Implementierung zu Kapitel 4 formalisiert und automatisiert geprüft wurden. Für das Verständnis und die Nachvollziehbarkeit werden hier nur Beweisideen präsentiert.

3.1 Prinzipale Typen

Betrachten wir wie in Abschnitt 2.3.2 den Term **I** mit den dort vorgestellten Typen $a \to a$ und $(a \to b) \to a \to b$. Wir wissen, dass der Term **I** beide Typen inhabitiert. Wenn wir beide Typen miteinander vergleichen, fällt auf, dass $(a \to b) \to a \to b$ die Einschränkung vorgibt, dass der Term **I** einen Term mit Funktionstypen $(a \to b)$ als Eingabe erhält, wohingegen $a \to a$ keine solche Einschränkung vorgibt. Genauer können wir sagen, dass $a \to a$ ein allgemeinerer Typ als $(a \to b) \to a \to b$ ist.

Um die Idee des allgemeineren Typs zu formalisieren, nutzen wir die Typsubstitution (siehe Definition 2.35). Wenn wir mittels der Typsubstitution alle Vorkommen eines Typatoms durch einen anderen Typen ersetzen, konkretisieren wir damit den Ursprungstypen und schränken diesen ein. In unserem Beispiel wäre die Typsubstitution mit $Su(a) = (a \to b)$ gegeben.

Definition 3.1: Allgemeinerer Typ

Falls es eine Substitution Su gibt, sodass $Su(\rho') = \rho$ gilt, nennen wir ρ' einen allgemeineren Typen als ρ und ρ einen Subtypen von ρ'.
Fall es weiter keine Substitution Su^{-1} gibt, sodass $Su^{-1}(\rho) = \rho'$, nennen wir ρ' einen echt allgemeineren Typen als ρ und ρ einen echten Subtypen von ρ'.

Notation

Falls ρ ein Subtyp von ρ' ist, schreiben wir auch $\rho \preceq \rho'$. Falls ρ ein echter Subtyp von ρ' ist, schreiben wir auch $\rho \prec \rho'$.

Bemerkung: Der Begriff des Subtyps ist nicht mit dem Begriff des Teiltyps (siehe Definition 2.25) zu verwechseln.

Lemma 3.2

Die Relation $\preceq$ ist eine transitiv-reflexive Ordnung.

Beweis:

Reflexiv: Wir wählen die Identität als Substitution.

Transitiv: Sei $\rho_1 \preceq \rho_2$ gegeben durch $Su_1(\rho_2) = \rho_1$ und $\rho_2 \preceq \rho_3$, gegeben durch $Su_2(\rho_3) = \rho_2$. Es gibt nun $Su_3 = Su_1 \circ Su_2$ mit $Su_3(\rho_3) = Su_1(Su_2(\rho_3)) = Su_1(\rho_2) = \rho_1$, also $\rho_1 \prec \rho_3$.

Wir betrachten nun, was es bedeutet, dass ein Typ ein allgemeinster Typ zu einem Term ist.

Definition 3.3: Prinzipaler Typ

Wir nennen einen Typen ρ den *allgemeinsten* oder *prinzipalen* Typen eines λ-Terms M, wenn gilt $\vdash M : \rho$ und für jeden Typen ρ' mit $\vdash M : \rho'$ es eine Substitution Su gibt, sodass $Su(\rho) = \rho'$.

Damit eine Substitution Su eine Umkehrsubstitution Su^{-1} hat, muss Su eine Umbenennung gewesen sein, da ansonsten ein atomarer Typ auf einen Funktionstypen abgebildet wurde, was durch Substitution nicht umkehrbar ist. Wir können daraus direkt folgern, dass es für eine Substitution Su nur zwei Möglichkeiten gibt, damit es keine entsprechende Umkehrsubstitution Su^{-1} gibt.

Lemma 3.4

Falls $Su(\rho) = \rho'$, aber es keine Substitution Su' mit $Su'(\rho') = \rho$ gibt, muss Su mindestens eine der beiden Eigenschaften erfüllen.

1. Su bildet zwei verschiedene Typatome a, b auf denselben Typen ϑ ab.
2. Su bildet einen atomaren Typen a auf einen Funktionstypen ab.

Beweis: Eine Typsubstitution Su bildet von atomaren Typen in einfache Typen ab. Die folgende Aufteilung von Substitutionen ist somit erschöpfend.

1. Su bildet alle Typatome auf verschiedene Typatome ab.
2. Su bildet zwei verschiedene Typatome a, b auf denselben Typen ϑ ab.
3. Su bildet einen atomaren Typan a auf einen Funktionstypen ab.

Im ersten Fall gibt es eine Umkehrsubstitution Su^{-1}.

Beispiel 3.5

$a \to a$ ist der prinzipale Typ von **I**.

Beweis: Zunächst können wir feststellen, dass $\vdash \mathbf{I} : a \to a$. Nehmen wir nun an, dass es einen Typen ρ mit $\vdash \mathbf{I} : \rho$ gibt, sodass $Su(\rho) = a \to a$, aber es keine Substitution Su^{-1} mit $Su^{-1}(a \to a) = \rho$ gibt. Nach Lemma 3.4 muss ρ entweder ein Pfeiltyp $b \to c$ oder b sein. Beide Typen werden jedoch nicht von **I** inhabitiert.

Definition 3.6: Prinzipaler Inhabitant

Wenn $\vdash M : \rho$ und ρ der prinzipale Typ von M ist, dann nennen wir M einen *prinzipalen Inhabitanten* von ρ.

Prinzipalität, so wie wir sie definiert haben, bleibt nicht unter β-Reduktion erhalten. Betrachten wie hierfür den Typen $a \to a \to a$. Der Term hat den prinzipalen Inhabitanten $(\lambda x\ y\ z.\mathbf{K}(x\ y)\ (x\ z))\ \mathbf{I}$[14, S. 177]. Evaluieren wir den Term durch die β-Reduktion, erhalten wir den Term $\lambda y\ z.\mathbf{K}\ y\ z$. Wir können zeigen, dass $\vdash \lambda y\ z.\mathbf{K}\ y\ z : a \to b \to a$, aber es gibt keine Substitution Su, sodass $Su(a \to a \to a) = a \to b \to a$. Damit ist $a \to a \to a$ nicht der prinzipale Typ von $\lambda y\ z.\mathbf{K}\ y\ z$. Dies hängt damit zusammen,

dass Redexe innerhalb eines Termes zusätzliche Einschränkungen an einen Typen einführen, die durch die β-Reduktion aufgelöst werden. Diese Beobachtung ist eng verwandt mit der Bemerkung zu Lemma 2.34.

Wir betrachten im Folgenden nur prinzipale Inhabitanten in β-Normalform, so genannte normale prinzipale Inhabitanten.

Definition 3.7: Normaler prinzipaler Inhabitant

Wir nennen einen λ-Term M einen *normalen prinzipalen Inhabitanten* eines Typs ρ, wenn M in β-Normalform ist und ρ der prinzipale Typ von M ist.

Das folgende Lemma erlaubt es uns, auf prinzipale Inhabitanten in η-langer Normalform zu beschränken.

Lemma 3.8: [14, 8A11.2]

Falls ein Term M in β-Normalform den prinzipalen Typen ρ hat, so hat auch seine eindeutige η-Expansion $M^+ \in \mathrm{Long}(\rho)$ den prinzipalen Typ ρ.

Beweis: Siehe [14, 8A11.2].

Analog zu Abschnitt 2.3.3 lässt sich ein Inhabitationsbegriff für prinzipale Typen und Terme in β-Normalform formulieren.

Definition 3.9: Prinzipales Inhabitationproblem

Gegegen ein einfacher Typ ρ, gibt es einen λ-Term M in β-Normalform, sodass ρ der prinzipale Typ von M ist?

[12] zeigt, dass das Problem PSPACE-vollständig ist. Wir werden hier zwei Konstruktionen, die in der Arbeit für den Beweis der Vollständigkeit genutzt werden, zunächst erarbeiten und in Kapitel 4 in Coq formalisieren. Somit wird eine Grundlage geschafft, um die PSPACE-Vollständigkeit zu verifizieren und die Vermutung, die am Ende der von [12] aufgestellt wurde, zu beweisen. Diese Arbeit wird jedoch nur die Verifikation der Grundlagen behandeln.

$$\dfrac{\dfrac{\dfrac{}{x : a, y : b \to c \vdash x : a}\,(\text{var})}{x : a \vdash \lambda y.x : (b \to c) \to a}\,(\text{abs})}{\vdash \lambda x\ y.x : a \to (b \to c) \to a}\,(\text{abs})$$

Abbildung 3.1: Typableitung für $\vdash \mathbf{K} : a \to (b \to c) \to a$

3.2 Teilformelfiltration

Wir werden nun eine Methodik vorstellen, mit der wir ein notwendiges Kriterium für die Prinzipalität formulieren und beweisen können. Ursprünglich in [13] für Intersektionstypen entwickelt, wurde die Teilformelfiltration (*Subformula Filtration*) in [12] für die einfachen Typen adaptiert.

Betrachten wir zunächst den Term $\mathbf{K} = \lambda x\ y.x$, seinen Typ $a \to (b \to c) \to a$ und explizit die Typableitung für $\vdash \mathbf{K} : a \to (b \to c) \to a$ in Abbildung 3.1. Es fällt auf, dass der Typ $b \to c$ für y nur durch die Abstraktion erzwungen wird, aber kein Teilterm von $\mathbf{K}$ mit $b \to c$ getypt wird. Wir könnten somit genau diese eine Instanz von $b \to c$ durch einen beliebigen Typen ersetzen und könnten $\mathbf{K}$ immer noch mit dem resultierendem Typen typen. Ersetzen wir hingegen genau ein a durch einen beliebigen anderen Typen, können wir $\mathbf{K}$ nicht mehr mit dem Ergebnistypen typen.

Bemerkung: Wenn wir den Typen $a \to (b \to c) \to a$ betrachten, könnten wir zu der Vermutung kommen, dass $b \to c$ beliebig wählbar ist, da es einzeln in dem Typen vorkommt. Dies ist nicht der Fall. Betrachten wir alternativ den Typen $a \to a \to a$, der ebenfalls von $\mathbf{K}$ inhabitiert wird. In diesem Beispiel können wir das mittlere a beliebig ersetzen, obwohl a mehrfach in dem Typen vorkommt.

Diese Beobachtungen lassen sich als Begriff der relevanten Typen formalisieren.

Definition 3.10: Relevante Typen

Gegeben eine Typableitung $\mathcal{D}$, dann definieren wir die Menge $T(\mathcal{D})$ an *relevanten Typen* in $\mathcal{D}$ folgendermaßen.

$$T(\mathcal{D}) = \{\rho \mid \Gamma \vdash M : \rho \text{ kommt in } \mathcal{D} \text{ vor}\}$$

Beispiel 3.11

Sei $\mathcal{D}$ die Typableitung in Abbildung 3.1, dann ist $T(\mathcal{D}) = \{a, (b \to c) \to a, a \to (b \to c) \to a\}$.

Intuitiv ist klar, dass, wenn $\mathcal{D} \rhd \emptyset \vdash M : \rho$ gilt und wir alle irrelevanten Typen $\rho \notin T(\mathcal{D})$, die als Teiltyp von ρ vorkommen, durch einen beliebigen anderen Typen ersetzen, wir eine Typableitung $\mathcal{D}'$ finden, sodass $\mathcal{D}'$ den Term M mit dem resultierenden Typen typt, da wir nur irrelevante Typen ersetzt haben. Um diese Intuition zu formalisieren, formulieren wir zunächst eine allgemeine Funktion, die in einem Typen alle Teiltypen, die für eine gegebene Menge irrelevant sind, durch einen neuen Typen ersetzt.

Definition 3.12: Filtratrionsfunktion

Sei X eine Menge an Typen und $a \in \mathbb{A}$ ein Typatom, dann definieren wir die *Filtrationsfunktion* $\mathcal{F}^a_X$ wie folgt:

$$\begin{aligned} \mathcal{F}^a_X(b) &= a && \text{falls } b \in \mathbb{A} \\ \mathcal{F}^a_X(\sigma \to \tau) &= \begin{cases} \mathcal{F}^a_X(\sigma) \to \mathcal{F}^a_X(\tau) & \text{falls } \sigma \to \tau \in X \text{ und } \tau \in X \\ a & \text{ansonsten} \end{cases} \end{aligned}$$

Bemerkung:

- Analog zu der Typsubstitution können wir die Filtration über Typkontexte fortsetzen. Auch hier schreiben wir $\mathcal{F}^a_X(\Gamma)$, da anhand der Eingabe immer eindeutig ist, ob es sich um die Filtration oder um die Fortsetzung auf Typkontexte handelt.
- Trotz Ähnlichkeit ist die Filtration *keine* Substitution nach Definition 2.35. Während die Filtration es erlaubt, dass ein Funktionstyp durch ein Typatom ersetzt wird, kann dies durch eine Substitution nicht geschehen.

Wir stellen fest, dass in Beispiel 3.11 nicht nur der Typ a relevant ist, sondern auch Typen, deren Teiltyp ein nicht relevanter Typ ist. Hierfür können wir den Typ $(b \to c) \to a \in T(\mathcal{D})$ betrachten, der relevant ist, wohingegen $b, c \notin T(\mathcal{D})$. In diesem Fall ist die Struktur des Typs relevant, also dass es sich um einen Funktionstypen, $\sigma \to a$ für irgendein σ, handelt. Typen, die für eine Typableitung $\mathcal{D}$ relevant sind, können somit irrelevante Teiltypen haben. Wichtig ist hier, dass dies weder die Teiltypen

relevant, noch den Gesamttypen irrelevant macht. Unsere Filtrationsfunktion ersetzt somit nicht den kompletten Funktionstypen, sondern nur das $b \to c$ in $(b \to c) \to a$, da $(b \to c) \notin T(\mathcal{D})$.

Beispiel 3.13

Sei $\mathcal{D}$ die Ableitung in Abbildung 3.1, dann ist

$$\begin{aligned}\mathcal{F}^d_{T(\mathcal{D})}(a \to (b \to c) \to a) &= \mathcal{F}^d_{T(\mathcal{D})}(a) \to \mathcal{F}^d_{T(\mathcal{D})}((b \to c) \to a) \\ &= a \to \mathcal{F}^d_{T(\mathcal{D})}(b \to c) \to \mathcal{F}^d_{T(\mathcal{D})}(a) \\ &= a \to d \to a.\end{aligned}$$

Wir können nun unsere Intuition formalisieren, dass wir irrelevante Typen ersetzen können, dabei aber einen Typen erhalten, der den Ursprungsterm immer noch inhabitiert. Wir stellen sogar die verallgemeinerte Aussage auf, dass die relevanten Typen nur eine Teilmenge der nicht zu ersetzenden Typen sein muss. Zunächst setzen die Filtrationsfunktion auf Typableitungen fort. Auch hierbei nennen wir die Funktion $\mathcal{F}^a_X$, da aufgrund der Eingabe eindeutig ist, welche Ausprägung der Funktion $\mathcal{F}^a_X$ gemeint ist.

Definition 3.14

Sei $a \in \mathbb{A}$ ein Typatom, X eine Menge an Typen, $M \in \Lambda$ ein λ-Term, $\rho \in \mathbb{T}$ ein einfacher Typ, Γ ein Typkontext und $\mathcal{D}$ eine Typableitung, sodass $\mathcal{D} \rhd \Gamma \vdash M : \rho$. Wir definieren die Fortsetzung einer Filtration $\mathcal{F}^a_X$ auf Typableitungen folgendermaßen.

$$\mathcal{F}^a_X(\mathcal{D}) = \left\{ \begin{array}{l} \dfrac{\mathcal{F}^a_X(\Gamma_1) \vdash M_1 : \mathcal{F}^a_X(\rho_1) \ldots \mathcal{F}^a_X(\Gamma_n) \vdash M_n : \mathcal{F}^a_X(\rho_n)}{\mathcal{F}^a_X(\Gamma') \vdash M' : \mathcal{F}^a_X(\rho')}\ (X) \\ \quad \Big| \quad \dfrac{\Gamma_1 \vdash M_1 : \rho_1 \ldots \Gamma_n \vdash M_n : \rho_n}{\Gamma' \vdash M' : \rho'}\ (X) \in \mathcal{D} \end{array} \right\}$$

Bemerkung: Wir interpretieren hier den Ableitungsbaum als Menge von angewandten Inferenzregeln, sodass $\mathcal{D} \rhd \Gamma \vdash M : \rho$ genau dann gilt, wenn es eine Regel $\frac{X_1 \ldots X_n}{\Gamma \vdash M : \rho} \in \mathcal{D}$ gibt und alle Wege an Inferenzregeln in Inferenzregeln ohne Hypothese enden.

Das Ergebnis einer Filtration erfüllt nicht zwangsweise die Bedingung an eine korrekte Typableitung, Lemma 3.15 wird aber zeigen, dass durch die Filtration eine gültige Ableitung entsteht.

Lemma 3.15

Falls $\mathcal{D} \rhd \Gamma \vdash M : \rho$ und $T(\mathcal{D}) \subseteq X$, dann gilt für jedes $a \in \mathbb{A}$, $\mathcal{F}^a_X(\mathcal{D}) \rhd \mathcal{F}^a_X(\Gamma) \vdash M : \mathcal{F}^a_X(\rho)$.

Beweis: Wir können eine Induktion über die Ableitung $\mathcal{D}$ durchführen.

Induktionsanfang: Es gilt

$$\mathcal{D} = \left\{ \frac{}{x : \rho \vdash x : \rho} \text{ (var)} \right\},$$

sowie

$$\mathcal{F}^a_X(\mathcal{D}) = \left\{ \frac{}{x : \mathcal{F}^a_X(\rho) \vdash x : \mathcal{F}^a_X(\rho)} \text{ (var)} \right\}$$

und damit $\mathcal{F}^a_X(\mathcal{D}) \rhd x : \mathcal{F}^a_X \vdash x : \mathcal{F}^a_X(\rho)$.

Fall (abs): Es gilt $\mathcal{D} \rhd \Gamma \vdash \lambda x.M : \sigma \to \tau$ und zu zeigen ist $\mathcal{F}^a_X(\mathcal{D}) \rhd \mathcal{F}^a_X(\Gamma) \vdash \lambda x.M : \mathcal{F}^a_X(\sigma \to \tau)$. Unsere Induktionshypothese sagt, dass $\mathcal{F}^a_X(\Gamma, x : \sigma) \vdash M : \mathcal{F}^a_X(\tau)$ und der Ableitungsbaum ist die Filtration über $\mathcal{D}$ ohne den letzten Schritt.

Es gilt somit

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau}(\text{abs}) \ \in \mathcal{D},$$

sowie für $\tau, \sigma \to \tau \in T(\mathcal{D}) \subseteq X$, dass

$$\mathcal{F}^a_X(\sigma \to \tau) = \mathcal{F}^a_X(\sigma) \to \mathcal{F}^a_X(\tau)$$

und

$$\mathcal{F}^a_X(\Gamma, x : \sigma) = \mathcal{F}^a_X(\Gamma), x : \mathcal{F}^a_X(\sigma).$$

Wir wissen dann, dass

$$\frac{\mathcal{F}^a_X(\Gamma), x : \mathcal{F}^a_X(\sigma) \vdash M : \mathcal{F}^a_X(\tau)}{\mathcal{F}^a_X(\Gamma) \vdash \lambda x.M : \mathcal{F}^a_X(\sigma) \to \mathcal{F}^a_X(\tau)}(\text{var}) \ \in \mathcal{F}^a_X(\mathcal{D})$$

und dies ist eine gültige (var)-Regel, deren Hypothese über unsere Induktionshypothese bewiesen wird.

Lemma 3.15 (Fortsetzung)

Fall (app) Es gilt $\mathcal{D} \rhd \Gamma \vdash M\ N : \tau$ und zu zeigen ist $\mathcal{F}^a_X(\mathcal{D}) \rhd \mathcal{F}^a_X(\Gamma) \vdash M\ N : \mathcal{F}^a_X(\tau)$. Wir haben zwei Induktionshypothesen, $\mathcal{F}^a_X(\Gamma) \vdash M : \mathcal{F}^a_X(\sigma \to \tau)$ und $\mathcal{F}^a_X(\Gamma) \vdash N : \mathcal{F}^a_X(\sigma)$, die jeweils von der Filtration der beiden Teilbäume in $\mathcal{D}$ bewiesen werden.

Es gilt somit

$$\frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash M\ N : \tau}(\text{app}) \ \in \mathcal{D},$$

sowie für $\sigma, \tau, \sigma \to \tau \in T(\mathcal{D}) \subseteq X$, dass

$$\mathcal{F}^a_X(\sigma \to \tau) = \mathcal{F}^a_X(\sigma) \to \mathcal{F}^a_X(\tau).$$

Wir wissen somit, dass

$$\frac{\mathcal{F}^a_X(\Gamma) \vdash M : \mathcal{F}^a_X(\sigma) \to \mathcal{F}^a_X(\tau) \qquad \mathcal{F}^a_X(\Gamma) \vdash N : \mathcal{F}^a_X(\sigma)}{\mathcal{F}^a_X(\Gamma) \vdash M\ N : \mathcal{F}^a_X(\tau)}(\text{app}) \ \in \mathcal{F}^a_X(\mathcal{D})$$

und dies ist eine gültige (app)-Regel, deren Hypothesen über die beiden Induktionshypothesen bewiesen werden.

Bemerkung: In [12] wurde eine Einschränkung von Lemma 3.15 gezeigt, die erforderte, dass a frisch ist. Das Lemma gilt sogar für beliebige a. Des Weiteren wurde in [12] der resultierende Baum nicht explizit konstruiert, sondern nur bewiesen, dass der Term mit dem filtrierten Typen ableitbar ist. Die hier vorgestellte Erweiterung erlaubt es uns direkt den resultierenden Beweisbaum anzugeben.
Dieser Beweis ist hier ausführlicher dargelegt, da wir in Kapitel 4 nur die Existenz eines Ableitungsbaumes beweisen. Dies reicht für die weitere Behandlung aus.

Beispiel 3.16

Sei $\mathcal{D}$ erneut die Ableitung aus Abbildung 3.1, dann ist $\mathcal{F}^d_{T(\mathcal{D})}(\mathcal{D})$ der folgende Beweisbaum.

$$\dfrac{\dfrac{\dfrac{}{x : a, y : d \vdash x : a}\,(\text{var})}{x : a \vdash \lambda y.x : d \to a}\,(\text{abs})}{\vdash \lambda x\ y.x : a \to d \to a}\,(\text{abs})$$

Die Filtration hat in Beispiel 3.16 den irrelevanten Teiltypen $b \to c$ ersetzt und hat gezeigt, dass der zweite Parameter von **K** nicht zwangsläufig eine Funktion sein muss, sondern ein beliebiger Typ d sein kann.

Der resultierende Typ $a \to d \to a$ ist insbesondere der prinzipale Typ von **K**, dies gilt aber nicht allgemein. Es ist sogar so, dass für eine Typableitung $\mathcal{D} \rhd \Gamma \vdash M : \rho$ und $T(\mathcal{D}) \subseteq X$ nicht zwangsweise gilt, dass $\rho \preceq \mathcal{F}^a_X(\rho)$, selbst, wenn a ein frisches Typatom ist, wie Beispiel 3.17 zeigt.

Beispiel 3.17

Sei $\mathcal{D}$ eine Typableitung mit $\mathcal{D} \rhd \Gamma \vdash \lambda x\ y\ z.x : a \to (b \to c) \to (c \to d) \to a$, dann ist $\mathcal{F}^e_{T(\mathcal{D})}(a \to (b \to c) \to (c \to d) \to d) = a \to e \to e \to a$, aber $a \to (b \to c) \to (c \to d) \to a \not\preceq a \to e \to e \to a$, da e entweder zu $(b \to c)$ oder zu $(c \to d)$ substituiert werden kann, aber nicht zu beiden.

Dass dennoch ein starker Zusammenhang zwischen filtrierten Typen und Prinzipalität existiert, zeigt das folgende Lemma 3.18, in dem wir über die Filtration ein notwendiges Kriterium für die Prinzipalität zeigen.

Lemma 3.18

Falls $\mathcal{D} \rhd \emptyset \vdash M : \rho$ und ρ einen Teiltypen $\sigma \to \tau$ enthält, sodass $\tau \notin T(\mathcal{D})$, dann hat M nicht den prinzipalen Typen ρ.

Beweis: Wir zeigen das Lemma durch Widerspruch. Nehmen wir zunächst an, dass M den prinzipalen Typen ρ hat. Betrachten wir die Typableitung $\mathcal{D}$. Grundsätzlich wissen wir über Lemma 3.15, dass eine Filtration mit von $\mathcal{D}$ mit $F^a_{T(\mathcal{D})}$ ebenfalls eine gültige Typableitung ist. Es gilt $\mathcal{F}^a_{T(\mathcal{D})}(\mathcal{D}) \rhd \emptyset \vdash M : \mathcal{F}^a_{T(\mathcal{D})}(\rho)$. Da $\tau \notin T(\mathcal{D})$ gilt, gibt es in $F^a_{T\mathcal{D}}(\rho)$ an der Stelle, an der in ρ sich der Teiltyp $\sigma \to \tau$ befand, entweder keinen Teiltypen, da ein größerer Teiltyp von ρ bereits auf a abgebildet wurde, oder $\sigma \to \tau$ selbst wurde auf a abgebildet. Es gibt somit es keine Substitution Su, sodass $Su(\rho) = \mathcal{F}^a_X(\rho)$, da Substitutionen *größere* Typen nicht auf kleinere Typen abbilden können. Siehe hierfür auch die Bemerkung zu Definition 3.12.

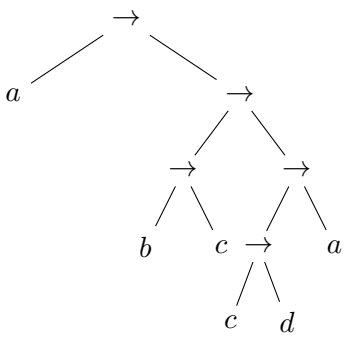


Abbildung 3.2: Der Baum zu $a \to (b \to c) \to (c \to d) \to a$

3.3 Teilformelkalkül

Die Teilformelfiltration erlaubt es uns, unnötige Strukturen innerhalb eines Typen zu eliminieren. Es ist klar, dass ein prinzipaler Typ keine unnötigen Strukturen mehr beinhaltet. Betrachten wir aber Beispiel 3.17, sehen wir, dass dies nicht ausreicht, um prinzipale Typen zu erhalten. In dem Beispiel wurden sowohl y und z mit e getypt. Damit verlangt der resultierende Typ, dass diese Eingaben denselben Typen haben müssen. Dies muss offensichtlich nicht gelten.

In diesem Abschnitt werden wir einen Kalkül kennenlernen, mithilfe dessen wir Teiltypen innerhalb eines Typen in Relation setzen können, um anzugeben, dass sie denselben Typen haben müssen. Hierfür benötigen wir eine Methode, um Teiltypen eines Typen zu adressieren. Betrachten wie dazu den Typen $\rho = a \to (b \to c) \to (c \to d) \to a$. Die Menge der Teiltypen von ρ ist $\{a \to (b \to c) \to (c \to d) \to a, (b \to c) \to (c \to d) \to a, (c \to d) \to a, b \to c, c \to d, a, b, c, d\}$. Es fällt auf, dass obwohl a und c zweimal in ρ enthalten sind, diese jeweils nur einmal als Teiltyp in der Menge vorkommen. Um explizit verschiedene Instanzen von a bzw. c innerhalb ρ zu referenzieren führen wir *Pfade* in Typen ein. Grundsätzlich kann jeder einfache Typ als binärer Baum interpretiert werden, dessen innere Knoten mit $\to$ und dessen Blätter mit Typatomen beschriftet sind. Der Typ $a \to (b \to c) \to (c \to d) \to a$ entspricht dem Baum in Abbildung 3.2.

Um nun zwischen den beiden a unterscheiden zu können, können wir angeben, welchen Weg wir im Baum gehen. Das erste a erreichen wir, indem wir einmal links gehen, das zweite a, indem wir dreimal rechts gehen. Dies lässt sich durch Definition 3.19 formalisieren.

Definition 3.19: Pfad im Typ

Wir definieren Pfade in einem Typen als Strings über dem Alphabet $\{\texttt{Src}, \texttt{Tgt}\}$. Wir bezeichnen die Menge an aller Pfade mit $\Pi = \{\texttt{Src}, \texttt{Tgt}\}^*$.
Jeder Typ ρ induziert eine partielle Funktion $P_\rho : \Pi \nrightarrow \mathbb{T}$ mit

$$P_\rho(\epsilon) = \rho$$
$$P_{\sigma\to\tau}(\texttt{Src} \cdot \pi) = P_\sigma(\pi)$$
$$P_{\sigma\to\tau}(\texttt{Tgt} \cdot \pi) = P_\tau(\pi).$$

$\mathrm{dom}(P_\rho)$ ist somit die Menge aller möglichen Pfade in ρ und $\mathrm{codom}(P_\rho)$ ist die Menge aller Teiltypen in ρ. Siehe hierfür auch Definition 2.25.

Notation

Wir schreiben abkürzend $\texttt{Src}^n$ bzw. $\texttt{Tgt}^n$, anstelle von $\underbrace{\texttt{Src} \cdot \ldots \cdot \texttt{Src}}_{n-\text{mal}}$ bzw. $\underbrace{\texttt{Tgt} \cdot \ldots \cdot \texttt{Tgt}}_{n-\text{mal}}$.

Beispiel 3.20

Sei $\rho = a \to (b \to c) \to (c \to d) \to a$, dann ist $P_\rho(\texttt{Src}) = a$ und $P_\rho(\texttt{Tgt} \cdot \texttt{Tgt} \cdot \texttt{Tgt}) = a$ und $\mathrm{dom}(P_\rho) = \{\epsilon, \texttt{Src}, \texttt{Tgt}, \texttt{Tgt} \cdot \texttt{Src}, \texttt{Tgt} \cdot \texttt{Src} \cdot \texttt{Src}, \texttt{Tgt} \cdot \texttt{Src} \cdot \texttt{Tgt}, \texttt{Tgt}\cdot\texttt{Tgt}, \texttt{Tgt}\cdot\texttt{Tgt}\cdot\texttt{Src}, \texttt{Tgt}\cdot\texttt{Tgt}\cdot\texttt{Src}\cdot\texttt{Src}, \texttt{Tgt}\cdot\texttt{Tgt}\cdot\texttt{Src}\cdot\texttt{Tgt}, \texttt{Tgt}\cdot\texttt{Tgt}\cdot\texttt{Tgt}\}$

Mittels der Pfade können wir nun angeben, welche Teiltypen innerhalb des filtrierten Typen $\rho = a \to e \to e \to a$ aus Beispiel 3.17 für den Term $\lambda x\ y\ z.x$ zusammengehören. Es sind nur die Typen, die zu der Variable x gehören, also $\texttt{Src}$ und $\texttt{Tgt} \cdot \texttt{Tgt} \cdot \texttt{Tgt}$. Wir können hier auch sehen, dass dies auch in dem ungefilterten Typen $a \to (b \to c) \to (c \to d) \to a$ gilt, da Teiltypen von irrelevanten Typen zu keinem Teilterm des λ-Terms gehören.

Betrachten wir hier einmal den prinzipalen Typen $a \to b \to c \to a$ des Terms $\lambda x\ y\ z.x$, so stellen wir fest, dass genau die Pfade zusammengehören, die zu identischen Typen führen. Aufbauend auf dieser Beobachtung werden wir zunächst ein weiteres notwendiges Kriterium und danach ein hinreichendes Kriterium für die Prinzipalität aufstellen.

Zunächst geben wir einen Kalkül an, das λ-Terme mit Pfaden in Relation setzt. Dies wird uns eine Relation zwischen Pfaden liefern, die angibt, welche Teiltypen eines Typen für einen Term zusammengehören, also identisch sein müssen.

Definition 3.21: Teilformelkalkül

Sei $\Delta \subseteq \Lambda \times \Pi$ ein Pfadkontext[a], $R \subseteq \Pi \times \Pi$ eine Relation zwischen Pfaden, $\pi, \pi' \in \Pi$ Pfade, $x \in \mathcal{V}$ eine Variable, und $M, M_1, \ldots, M_n \in \Lambda$ λ-Terme. Dann definieren wir die Relation $\_ \vdash_R \_ : \_$ anhand der folgenden Inferenzregeln.

$$\frac{\Delta, x : \pi \cdot \mathtt{Src} \vdash_R M : \pi \cdot \mathtt{Tgt}}{\Delta \vdash_R \lambda x.M : \pi} \ (\text{abs})_R$$

$$\frac{(\pi \cdot \mathtt{Tgt}^n, \pi') \in R \quad \Delta, x : \pi \vdash_R M_i : \pi \cdot \mathtt{Tgt}^{i-1} \text{ für } i = 1 \ldots n}{\Delta, x : \pi \vdash_R x\ M_1 \ldots M_n : \pi'} \ (\text{varapp})_R$$

Analog zu Definition 2.28 nennen wir den Baum mit Wurzel $\Delta \vdash M : \pi$ Pfadableitung und wenn $\mathcal{D}_R$ eine Pfadableitung ist, schreiben wir $\mathcal{D}_R \rhd \Delta \vdash M : \pi$.

[a] Pfadkontexte sind analog zu Typkontexten definiert.

Bemerkung:

- Wir nennen die Bedingungen in der Form $(\pi, \pi') \in R$ Nebenbedingungen. Auch wenn eine Relation R hier als Eingabe für den Teilformelkalkül definiert ist, können wir zeigen, dass R bestimmte Pfade in Relation setzen muss, damit ein Term M erfolgreich dem Pfad ε zugewiesen wird. R ist somit abhängig von M.
- Der Aufbau der Regeln von $\vdash_R$ ähnelt stark den Regeln (long) und (abs), die die η-langen Terme bestimmen. Diesen Zusammenhang werden wir uns in Lemma 3.32 zu Nutze machen.

Betrachten wir den Aufbau von $\vdash_R$, so wird klar, dass zusätzliche Einträge in R eine erfolgreiche Ableitung nicht fehlschlagen lassen. Dies bedeutet insbesondere, dass es für eine Ableitung $\vdash_R M : \varepsilon$ eine kleinste Menge $R' \subseteq R$ gibt, sodass $\vdash_{R'} M : \varepsilon$. Diese Menge ist für uns von Interesse, da sie genau die Pfade in Relation setzt, die in einem Typen, der M typt, gleich sein müssen.

Lemma 3.22

Falls $\vdash_R M : \varepsilon$ und $R \subseteq R'$, dann $\vdash_{R'} M : \varepsilon$

Beweis: Das einzige Vorkommen der Relation R ist in der $(\text{varapp})_R$ Regel in der Form $(\pi \cdot \texttt{Tgt}^n, \pi') \in R$. Es ist klar, dass aus $R \subseteq R'$ folgt, dass für alle $x \in R$, auch $x \in R'$ gilt.

Korollar 3.23

Falls $\vdash_R M : \varepsilon$, dann gibt es ein $R' \subseteq R$, sodass für jedes R'' mit $\vdash_{R''} M : \varepsilon$ gilt $R' \subseteq R''$.

Beweis: Aus Lemma 3.22 wissen wir, dass $\vdash_R$ monoton im Bezug auf die Teilmengenrelation zu R ist. Demnach gibt es eine kleinste Pfadrelation R', die Teilmenge jeder Pfadrelation R'' ist, die $\vdash R''$ für einen Term M erfüllt.

Das folgende Lemma und das darauf aufbauende Korollar geben uns Einschränkungen an eine minimale Relation R, also an die Tupel in R, die für die Ableitung notwendig sind.

Lemma 3.24

Falls $\mathcal{D}_R \rhd \emptyset \vdash_R M : \varepsilon$, dann gilt für jeden Inferenzschritt $\Delta \vdash_R N : \pi$ in $\mathcal{D}_R$,

(i) Die Anzahl an `Src` in π ist gerade.

(ii) Für jedes $(x : \pi') \in \Delta$ ist die Anzahl an `Src` in π' ungerade.

Beweisidee: Wir zeigen die allgemeinere Aussage, wenn $\mathcal{D}_R \rhd \Delta \vdash M : \pi$ und für π bzw. Δ die Aussagen (i) bzw. (ii) gelten, dann gelten sie in jedem Schritt in $\mathcal{D}$.

Fall $(\text{abs})_R$: Falls π eine gerade Anzahl an `Src` und in Δ alle Pfade eine ungerade Anzahl an `Src` haben, dann haben in $\Delta, x : \pi \cdot \texttt{Src}$ alle Pfade eine ungerade Anzahl an `Src`.

Fall $(\text{varapp})_R$: Falls in $\Delta, x : \pi$ alle Pfade eine ungerade Anzahl an `Src` haben, dann hat für alle i der Pfad $\pi \cdot \texttt{Tgt}^{i-1} \cdot \texttt{Src}$ eine gerade Anzahl an `Src`.

Sowohl (i) als auch (ii) gelten für $\vdash_R M : \varepsilon$.

Algorithmus 2 : R_M-aux

Data : Pfadkontext Δ, Pfad π', λ-Term M in β-Normalform

Result : Relation R_M oder Fehlschlag

```
1  switch M do
2      case M = λx.P do
3          return R_M-aux (Δ, x : π' · Src, (π' · Tgt), P)
4      case M = x M_1 ... M_n do
5          if x ∈ dom(Δ) then
6              π ← Δ(x)
7              R ← {(π · Src^n, π')}
8              for i ∈ [1...n] do
9                  R ← R ∪ R_m-aux(Δ, (π · Tgt^i · Src), M_i)
10             return R
11         else
12             fail
```

Korollar 3.25

Falls $\mathcal{D}_R \rhd \emptyset \vdash_R M : \varepsilon$, dann hat kein Inferenzschritt in $\mathcal{D}_R$ eine Nebenbedingung in der Form $(\pi, \pi) \in R$.

Beweisidee: Folgt direkt aus Lemma 3.24.

Bemerkung: Dies bedeutet insbesondere, dass einmal vorkommende Typen nicht mit sich selbst in Relation stehen.

Korollar 3.23 folgend werden wir nun für eine Ableitung $\vdash_R M : \epsilon$ eine von M abhängige minimale Relation R angeben. Hierfür betrachten wir den Algorithmus 2.

Bemerkung: Der Algorithmus kann fehlschlagen. Da der Algorithmus rekursiv definiert ist, führt ein Fehlschlag irgendeines Kindaufrufs zum Fehlschlag des Elternaufrufs.

Lemma 3.26

Wenn M ein geschlossener λ-Term in β-Normalform ist, dann schlägt R_M-aux$(\emptyset, \pi, M)$ nicht fehl.

Beweis: Algorithmus 2 schlägt nur fehl, wenn Kontext keinen Eintrag für eine benutzte Variable hat. Da beim Abarbeiten einer Abstraktion die Variable, über die abstrahiert wird, in den Kontext gelegt wird, und in einem geschlossenen Term über alle benutzten Variablen abstrahiert wird, schlägt Algorithmus 2 in diesem Fall nicht fehl.

Definition 3.27

Gegeben ein geschlossener λ-Term M in β-Normalform, dann ist R_M die kleinste transitiv-symmetrische Relation, sodass $\vdash_{R_M} M : \varepsilon$.

Bemerkung: In [12] wird an dieser Stelle ein Äquivalenzabschluss gefordert, wir kommen hier damit aus, nur Transitivität und Symmetrie zu fordern. Die Eigenschaft der Reflexivität führt insbesondere zu Problemen, wenn wir die Eigenschaft $(\star)$ (Definition 3.36) betrachten.
Nach Rücksprache mit Andrej Dudenhefner – einem der Autoren von [12] – stellt sich heraus, dass eine Art von *innerer* Reflexivität gemeint ist, sodass ein Pfad bereits in einem Tupel vorkommen muss, um im Abschluss vorhanden zu sein,

$$(a, b) \in R \Rightarrow (a, a) \in R \wedge (b, b) \in R.$$

Dies wird jedoch vollständig von dem transitiv-symmetrischen Abschluss überdeckt. Siehe hierzu den entsprechenden Beweis in Kapitel 4.

Notation

Wir schreiben R^{ts} für den transitiv-symmetrischen Abschluss der Relation R.

Lemma 3.28

R_M-aux$(\emptyset, \varepsilon, M)^{ts}$ aus Algorithmus 2 erfüllt Definition 3.27.

Lemma 3.28 (Fortsetzung)

Beweisidee: Der Algorithmus arbeitet die Inferenzregeln in Definition 3.21 rückwärts ab. An dieser Stelle wird auch auf eine genauere Beweisidee verzichtet und auf das Beispiel 3.29 verwiesen.

Beispiel 3.29

Sei $M = \lambda x\ y\ z.x$. Dann ist R_M der transitiv-symmetritische Abschluss von $R_M\text{-aux}(\emptyset, \varepsilon, M)$, und

$$\begin{aligned} R_M\text{-aux}(\emptyset, \varepsilon, M) &= R_M\text{-aux}(\{x : \mathtt{Src}\}, \mathtt{Tgt}, \lambda y\ z.x) \\ &= R_M\text{-aux}(\{x : \mathtt{Src}, y : \mathtt{Tgt} \cdot \mathtt{Src}\}, \mathtt{Tgt}^2, \lambda z.x) \\ &= R_M\text{-aux}(\{x : \mathtt{Src}, y : \mathtt{Tgt} \cdot \mathtt{Src}, z : \mathtt{Tgt}^2 \cdot \mathtt{Src}\}, \mathtt{Tgt}^3, x) \\ &= \{(\mathtt{Src}, \mathtt{Tgt} \cdot \mathtt{Tgt} \cdot \mathtt{Tgt})\} \\ &\quad \text{mit } n = 0, \pi \leftarrow \mathtt{Src}, R \leftarrow \{(\mathtt{Src}, \mathtt{Tgt} \cdot \mathtt{Tgt} \cdot \mathtt{Tgt})\}, \end{aligned}$$

und es gilt die folgende Ableitung:

$$\dfrac{\dfrac{\dfrac{\dfrac{(\mathtt{Src}, \mathtt{Tgt} \cdot \mathtt{Tgt} \cdot \mathtt{Tgt}) \in R_M}{x : \mathtt{Src}, y : \mathtt{Tgt} \cdot \mathtt{Src}, z : \mathtt{Tgt} \cdot \mathtt{Tgt} \cdot \mathtt{Tgt} \vdash_{R_M} x : \mathtt{Tgt} \cdot \mathtt{Tgt} \cdot \mathtt{Tgt}}\ (\text{varapp})_{R_M}}{x : \mathtt{Src}, y : \mathtt{Tgt} \cdot \mathtt{Src} \vdash_{R_M} \lambda z.x : \mathtt{Tgt} \cdot \mathtt{Tgt}}\ (\text{abs})_{R_M}}{x : \mathtt{Src} \vdash_{R_M} \lambda y\ z.x : \mathtt{Tgt}}\ (\text{abs})_{R_M}}{\vdash_{R_M} \lambda x\ y\ z.x : \varepsilon}\ (\text{abs})_{R_M}$$

Beispiel 3.29 zeigt, wie Algorithmus 2 und die Regeln aus Definition 3.21 entgegengesetzt laufen.

Wir haben nun einen Algorithmus, der anhand eines Terms über Pfade in Typen, die zu dem Term gehören, angibt, welche Positionen dort zusammengehören. Wir benötigen nun eine analoge Definition für Teiltypen in einem Typen. Da wir hier einen Typ direkt vorliegen haben, benötigt es keinen weiteren Kalkül und keinen Algorithmus, um zu entscheiden, welche Teiltypen innerhalb eines Typen identisch sind.

Definition 3.30

Gegeben ein Typ τ, dann ist die transitiv-symmetrische Relation R_τ wie folgt definiert.

$$R_\tau = \{(\pi, \pi') \mid \pi \neq \pi' \wedge P_\tau(\pi) = P_\tau(\pi') \in \mathbb{A}\}^{ts}$$

Bemerkung:

- Die Einschränkung $\pi \neq \pi'$ verhindert, dass Typen, die nur einmal in einem Typen vorkommen, in unsere Relation aufgenommen werden. Dies ist eine wichtige Einschränkung für $(\star)$.
- Auch hier verwenden wir analog zu Definition 3.27 und im Gegensatz zu [12] den transitiv-symmetrischen Abschluss, da auch hier die *innere* Reflexivität ausreicht. Die vollständige Reflexivität würde sogar im Bezug auf Definition 3.36 diese Bedingung zu einer Tautologie machen.

Beispiel 3.31

1. $R_{a\to e\to e\to a} = \{(\texttt{Src}, \texttt{Tgt}\cdot\texttt{Tgt}\cdot\texttt{Tgt}), (\texttt{Tgt}\cdot\texttt{Src}, \texttt{Tgt}\cdot\texttt{Tgt})\}^{ts}$
2. $R_{a\to(b\to c)\to(c\to d)\to a} = \{(\texttt{Src}, \texttt{Tgt}\cdot\texttt{Tgt}\cdot\texttt{Tgt}), (\texttt{Tgt}\cdot\texttt{Src}\cdot\texttt{Tgt}, \texttt{Tgt}\cdot\texttt{Tgt}\cdot\texttt{Src}\cdot\texttt{Src})\}^{ts}$
3. $R_{a\to b\to c\to a} = \{(\texttt{Src}, \texttt{Tgt}\cdot\texttt{Tgt}\cdot\texttt{Tgt})\}^{ts}$

Wir können beobachten, dass $a \to b \to c \to a$, der prinzipale Typ von $\lambda x\ y\ z.x$, die gleiche Pfadrelation induziert wie der Term $\lambda x\ y\ z.x$ selbst. Diese Beobachtung werden wir mit den nachfolgenden Lemmata formalisieren und zu einer notwendigen Bedingung für die prinzipale Inhabitation erweitern.

Betrachten wir die Ableitungsregeln in Definition 3.21 und Definition 2.40, so stellen wir eine strukturelle Ähnlichkeit fest. Beide Definitionen erlauben als λ-Terme entweder ein Abstraktion, oder eine Kette an Applikationen, die mit einer Variable beginnt. Wir nutzen nun diesen Zusammenhang, um R_M und R_τ in Relation zu setzen.

Lemma 3.32

Gegeben ein λ-Term M in β-Normalform und Typ τ, dann sind die folgenden Aussagen äquivalent.

1. $M \in \mathrm{Long}(\tau)$
2. $\vdash_{R_\tau} M : \varepsilon$
3. $R_M \subseteq R_\tau$

Beweisidee:

1 ⇒ 2: Nach Lemma 2.41 gilt $\vdash: M : \tau$. Daraus folgt, dass M ein geschlossener Term ist. Es gibt somit ein R_M, sodass $\vdash_{R_M} M : \varepsilon$. Aufgrund der Ähnlichkeit der Langableitung und des Teilformelkalküls können wir durch Induktion über die Langableitung feststellen, dass für jeden Schritt $\Delta \vdash_{R_M} M : \pi$ in der Pfadableitung es einen Schritt $\{x : P_\tau(\pi') \mid (x : \pi') \in \Delta\} \vdash M : P_\tau(\pi)$ in der Langableitung gibt.

Für einen Schritt in der Form

$$\frac{(\pi \cdot \mathtt{Tgt}^n, \pi') \in R_M \quad \Delta, x : \pi \vdash_{R_M} M_i : \pi \cdot \mathtt{Tgt}^{i-1} \text{ für } i = 1 \dots n}{\Delta, x : \pi \vdash_{R_M} x\ M_1\ \dots\ M_2 : \pi'}\ (\mathrm{varapp})_{R_M}$$

folgt aus der Langableitung, dass $P_\tau(\pi') \in \mathbb{A}$ und, da R_M die notwendigen Gleichheiten in den Typen für M beinhaltet, dass $P_\tau(\pi \cdot \mathtt{Tgt}^n) = P_\tau(\pi')$. Da weiterhin gilt, dass kein $(\pi, \pi) \in R_M$ als Nebenbedingung vorkommt, wissen wir, dass $\pi \cdot \mathtt{Tgt}^n \neq \pi'$. Damit erfüllt $(\pi \cdot \mathtt{Tgt}^n, \pi')$ alle Bedingungen für R_τ. Da alle verwendeten Tupel aus R_M in R_τ vorkommen, gilt auch $\vdash_{R_\tau} M : \varepsilon$.

2 ⇒ 3: R_M ist die minimale Relation R, die $\vdash_R M : \varepsilon$ erfüllt. Also ist sie Teilmenge aller dieser Relationen, insbesondere auch R_τ.

3 ⇒ 1: Nehmen wir an, dass $R_M \subseteq R_\tau$. Es ist klar, dass, falls R_M existiert, es eine Ableitung $\vdash_{R_M} M : \varepsilon$ gibt. Wir können die ähnliche Struktur der Pfad- und Langableitungen ausnutzen, um aus der Ableitung zu $\vdash_{R_M} M : \varepsilon$ eine Langableitung zu erhalten. Wir können $(\mathrm{varapp})_{R_M}$-Regeln übersetzen, da aus $(\pi \cdot \mathtt{Tgt}^2, \pi') \in R_\tau$ folgt, dass $P_\tau(\pi \cdot \mathtt{Tgt}^2) = P_\tau(\pi') \in \mathbb{A}$. Des Weiteren können wir $(\mathrm{abs})_{R_M}$-Regeln übersetzen, da falls $\pi \cdot \mathtt{Tgt} \in \mathrm{dom}(P_\tau)$, auch $\pi \cdot \mathtt{Src} \in \mathrm{dom}(P_\tau)$.

Lemma 3.32 (Fortsetzung)

Wir werden in Kapitel 4 genauer auf die Konstruktionen der einzelnen Schritte eingehen.

Bevor wir das notwendige Kriterium für die Prinzipalität beweisen können, muss ein weiteres Lemma aufgestellt werden, das es uns erlaubt für einem Term $M \in \text{Long}(\tau)$, atomare Typen zu ersetzen, sofern wir diese an allen Stellen ersetzen, die über R_M in Relation stehen.

Lemma 3.33

Gegeben ein λ-Term $M \in \text{Long}(\tau)$ und $\pi \in \text{dom}(P_\tau)$, sodass $P_\tau(\pi) = a \in \mathbb{A}$. Sei b ein frisches Atom, dann sei τ' definiert über den Typen τ, in dem jeder Teiltyp, der durch den Pfad $\pi' \in \text{dom}(P_\tau)$ mit $\pi = \pi'$ oder $(\pi, \pi') \in R_M$ bestimmt wird, durch b ersetzt wird. Dann ist $M \in \text{Long}(\tau')$.

Beweisidee: Falls $M \in \text{Long}(\tau)$ wissen wir über Lemma 3.32, dass $R_M \subseteq R_\tau$. Wenn wir in τ eine Variable a an Stelle π und an allen Stellen π', für die gilt $(\pi, \pi') \in R_M$, durch b ersetzen, dann ist auch $(\pi, \pi') \in R_{\tau'}$, da $P'_\tau(\pi) = P'_\tau(\pi') = b$. Damit gilt $R_M \subseteq R'_\tau$ und nach Lemma 3.32 auch $M \in \text{Long}(\tau')$.

Nun können wir die durch Beispiel 3.31 aufgestellte Intuition formalisiert, nach der für einen prinzipalen Typen τ zu einem Term M, die Menge R_τ und die Menge R_M identisch sein müssen. Die atomaren Teiltypen von τ erfüllen somit die durch die Pfade in R_M aufgestellte Gleichheit.

Lemma 3.34

Gegeben ein Typ τ, sei $M \in \text{Long}(\tau)$. Falls τ der prinzipale Typ von M ist, dann gilt $R_\tau = R_M$.

Beweisidee: Wir wissen, dass $R_\mathsf{M} \subseteq R_\tau$, da $M \in \text{Long}(\tau)$. Es bleibt zu zeigen, dass $R_\tau \subseteq R_M$.
Nehmen wir das Gegenteil an, nämlich dass es ein $(\pi, \pi') \in R_\tau$ gibt, welches nicht in R_M liegt. Wir wissen, dass $P_\tau(\pi) \in \mathbb{A}$. Wir wählen nun ein a, das ein frisches Typatom für τ ist, und konstruieren analog zu Lemma 3.33 ein τ', indem wir $P_\tau(\pi)$ durch a ersetzen. Da $P_\tau(\pi) \in \mathbb{A}$ gilt nach Lemma 3.33, dass

Lemma 3.34 (Fortsetzung)

$M \in \text{Long}(\tau')$. Da $(\pi, \pi') \notin R_M$, wurde der Typ an der Stelle π' nicht ersetzt. Es gibt somit keine Substitution von τ zu τ'. Damit kann τ nicht der prinzipale Typ sein.

Die Umkehrung, dass, falls $R_M = R_\tau$ gilt τ prinzipal zu M ist, gilt jedoch nicht zwangsläufig. Betrachten wir hierzu das folgende Beispiel.

Beispiel 3.35

Sei $M = \lambda x\, y.x$ und $\tau = a \to (b \to c) \to a$. Es gilt $R_M = \{(\texttt{Src}, \texttt{Tgt}\cdot\texttt{Tgt}\} = R_\tau$, aber τ ist nicht der prinzipale Typ von M. Genauer hat τ keinen prinzipalen Inhabitanten.

Ein Problem ist hierbei, dass wir zwar über R_M wissen, welche Typen in τ identisch sein müssen, aber keine Aussage über einzeln vorkommende Typen in τ treffen. Diese sind weder in R_M, noch in R_τ referenziert, und können demnach beliebig spezifisch sein.

Es ist klar, dass, wenn alle nicht durch R_M gebundene Teiltypen von τ atomare Typen sind und $R_M = R_\tau$, dann ist τ prinzipal zu M. Wir können diese Beobachtung sogar noch verallgemeinern, indem wir folgendes betrachten. Jeder nichtatomare Typ $\sigma' \to \tau'$ endet in einem Teiltypen der Form $\sigma \to a$, mit $a \in \mathbb{A}$. Weiter ist klar, dass, wenn ein Typatom a nur als Zieltyp in $\sigma \to a$ vorkommt, ein Term M mit diesem Typen niemals über eine Variable mit dem Typen a abstrahiert.

Wenn a nur einmal in τ vorkommt, wissen wir, dass es für jeden geschlossenen Term M, für den gilt $\vdash M : \tau$, keine Variable mit dem Typen a gibt. Da a auch im Resttypen nicht genutzt wird, wissen wir, dass innerhalb eines solchen Term M ein Teilterm vom Typen a niemals als Eingabe benutzt wird. a ist somit ein irrelevanter Typ für eine Ableitung $\vdash M : \tau$ nach Definition 3.10.

Die hier beschriebene Bedingung lässt sich als Bedingung $(\star)$ mit der folgenden Definition formalisieren.

Definition 3.36: $(\star)$

Wir sagen, ein Typ τ erfüllt $(\star)$, falls $\forall \pi \in \text{dom}(P_\tau).(P_\tau(\pi \cdot \texttt{Tgt}) \in \mathbb{A} \Rightarrow (\pi \cdot \texttt{Tgt}, \pi \cdot \texttt{Tgt}) \in R_\tau)$.

Mittels ($\star$) können wir nun eine notwendige Bedingung für die prinzipale Inhabitation formulieren.

Lemma 3.37

Falls τ nicht ($\star$) erfüllt, dann hat τ keine normalen prinzipalen Inhabitanten.

Beweis: Nehmen wir an, dass τ die Bedingung ($\star$) nicht erfüllt. Dann existiert ein Pfad $\pi \in \mathrm{dom}(\tau)$, sodass $P_\tau(\pi \cdot \mathtt{Tgt}) \in \mathbb{A}$, aber $(\pi \cdot \mathtt{Tgt}, \pi \cdot \mathtt{Tgt}) \notin R_\tau$. Nehmen wir weiter an, dass τ einen normalen prinzipalen Inhabitanten hat. Es gibt damit auch einen normalen prinzipalen Inhabitanten in η-langer Normalform $M \in \mathrm{Long}(\tau)$.
Nach Lemma 3.32 gibt es eine Ableitung $\mathcal{D}_{R_\tau} \rhd \emptyset \vdash_{R_\tau} M : \varepsilon$. Da $(\pi \cdot \mathtt{Tgt}, \pi \cdot \mathtt{Tgt}) \notin R_\tau$, kommt aufgrund der inneren Reflexivität in R_τ nirgendwo ein $\pi \cdot \mathtt{Tgt}$ vor. Somit gibt es in $\mathcal{D}_{R_\tau}$ keinen Schritt der Form $\Delta \vdash_{R_\tau} x\ M_1\ \ldots\ M_n : \pi \cdot \mathtt{Tgt}$ für irgendeinen Kontext Δ und Applikation $x\ M_1 \ldots M_n$.
Da $P_\tau(\pi \cdot \mathtt{Tgt}) \in \mathbb{A}$, liegt $\pi \cdot \mathtt{Tgt} \cdot \mathtt{Src}$ bzw. $\pi \cdot \mathtt{Tgt} \cdot \mathtt{Tgt}$ nicht in der Menge $\mathrm{dom}(P_\tau)$. Daraus folgt, dass kein Schritt $\Delta' \vdash_{R_M} \lambda x.P : \pi \cdot \mathtt{Tgt}$ für irgendein Δ' und irgendein $\lambda x.P$ in $\mathcal{D}_{R_\tau}$ vorkommt.
Übersetzt man die Ableitung zu einer Langableitung $\mathcal{D}' \rhd \emptyset \vdash M : \tau$, wissen wir, dass auch dort kein Schritt in der Form $\Gamma \vdash N : P_\tau(\pi \cdot \mathtt{Tgt})$ vorkommt. Somit enthält τ einen Teiltypen $P_\tau(\pi) = \sigma \to a$ für $P_\tau(\pi \cdot \mathtt{Src}) = \sigma$ und $P_\tau(\pi \cdot \mathtt{Tgt}) = a$, und $a \notin T(\mathcal{D})$. Damit ist nach Lemma 3.18 τ nicht der prinzipale Typ von M. Dies ist ein Widerspruch zu unserer Annahme.

Abschließend können wir die Bedingung ($\star$) nutzen, um eine hinreichende Bedingung für die Prinzipalität formulieren.

Lemma 3.38

Gegeben ein Typ τ, der ($\star$) erfüllt, sei $M \in \mathrm{Long}(\tau)$. Falls $R_\tau = R_{\mathsf{M}}$, dann ist τ der prinzipale Typ von M.

Beweis: Nehmen wir an, das τ nicht prinzipal für M ist, also es einen prinzipalen Typen τ' mit $\tau' \prec \tau$ gibt. Sei Su die Substitution, die τ' auf τ abbildet. Wir wissen durch Lemma 2.42, dass $M \in \mathrm{Long}(\tau')$ und über Lemma 3.34, dass $R_M = R_{\tau'}$. Wir zeigen nun, dass $R_\tau \neq R'_\tau$ und damit auch $R_\tau \neq R_M$. Nach Lemma 3.4 wissen wir, dass wir genau zwei Fälle für Su untersuchen müssen.

Lemma 3.38 (Fortsetzung)

Fall, Su ist eine Umbenennung: Da es keine Su^{-1} gibt, die τ auf τ' abbildet, muss Su zwei in τ' vorkommende, verschiedene Typatome auf dasselbe Typatom in τ abbilden. Also gibt es zwei Pfade π, π', sodass $P_{\tau'}(\pi) \in \mathbb{A} \neq P_{\tau'}(\pi') \in \mathbb{A}$, aber $P_\tau(\pi) = P_\tau(\pi') \in \mathbb{A}$. Es gilt somit $(\pi, \pi') \in R_\tau$, aber $(\pi, \pi') \notin R'_\tau$.

Fall, Su bildet ein Typatom auf einen Funktionstypen ab: Sei $\sigma_1 \to \ldots \to \sigma_n \to b$ mit $n > 0$ der Funktionstyp und $a \in \mathbb{A}$ das Typatom. Wenn $\pi \in \mathrm{dom}(\tau')$ mit $P_{\tau'}(\pi) = a$, dann ist auch $P_\tau(\pi) = \sigma_1 \to \ldots \sigma_n \to b$ und $P_\tau(\pi \cdot \mathtt{Tgt}^n) = b$. Da τ die Bedingung $(\star)$ erfüllt wissen wir, dass $(\pi \cdot \mathtt{Tgt}^n, \pi \cdot \mathtt{Tgt}^n) \in R_\tau$. Da $P_{\tau'}(\pi) \in \mathbb{A}$ wissen wir, dass $(\pi \cdot \mathtt{Tgt}^n, \pi \cdot \mathtt{Tgt}^n) \notin R_\tau$.

Der Vorteil der Charakterisierung der Prinzipalität über $(\star)$ ist, dass sich die Bedingung einfach und automatisiert prüfen lässt.

Wir können nun ein Theorem aufstellen, das zeigt, dass ein Typ genau dann prinzipal zu einem Term ist, wenn er $(\star)$ erfüllt und genau die Gleichheiten in seinen Typatomen erfüllt, die über den Aufbau des Terms vorgegeben sind.

Theorem 3.39

Gegeben ein Typ τ, der $(\star)$ erfüllt und ein λ-Term $M \in \mathrm{Long}(\tau)$, dann ist τ genau dann der prinzipale Typ von M, wenn $R_M = R_\tau$.

Beweis:

$\Rightarrow$ Lemma 3.34

$\Leftarrow$ Lemma 3.38

4 Verifikation durch Coq

Die Theorie des Teilformelkalküls und der Teilformelfiltration, die in dem letzten Kapitel vorgestellt wurden, wurde im Rahmen dieser Arbeit formalisiert und in Teilen mit einem computergestützten Beweishelfer verifiziert. Als System wurde hier Coq[1] in der Version 8.8.0 gewählt. Die hier vorgestellte Implementierung liegt einerseits der Arbeit bei und lässt sich unter `https://github.com/christofsteel/PrincInh/` abrufen. Diese Arbeit bezieht sich auf den Tag `final` (Commit `e1bdae8`). Die Implementierung steht unter der *Apache-2.0-Lizenz*[2].

Wir werden zunächst den Aufbau der grundlegenden Datentypen sowie den wichtigsten Lemmata zu diesen betrachten, um darauf aufbauend die Verifikation für Lemma 3.18 der Teilformelfiltration und Lemma 3.32 des Teilformelkalküls zu untersuchen.

4.1 Aufbau der Implementierung

Der grundlegende Aufbau der Implementierung ist über Tabelle 4.1 gegeben. Als externe Bibliothek wurde lediglich Autosubst verwendet [16]. Autosubst stellt Klassen bereit, über die Substitutionen definiert sind und generiert Lemmata, die den Umgang mit diesen Substitutionen vereinfachen. Da wir hier hauptsächlich Terme betrachten, die bereits in β-Normalform vorliegen, wurde dieser Teil nicht intensiv genutzt.

In den folgenden Abschnitten werden wir viele Listings betrachten; diese sind – falls vorhanden – immer mit der entsprechenden Quelltextdatei sowie der entsprechenden Zeilennummer annotiert. Der Quelltext musste an einigen Stellen für das Einbetten in diese Arbeit verändert werden. Dies betrifft jedoch lediglich Formatierungen. Dennoch können dadurch einige Zeilennummern nicht mehr mit den Zeilennummern im Originalquelltext übereinstimmen. Die Zeilennummer der ersten Zeile ist jedoch in jedem Fall korrekt. Die Implementierungen von Lemmata werden an dieser Stelle

[1] `https://coq.inria.fr/`

[2] `https://opensource.org/licenses/Apache-2.0`

Datei	Beschreibung
`Utils.v`	Grundlegende Hilfsfunktionen und Konstruktionen.
`Terms.v`	Definition von λ-Termen.
`Terms2.v`	Alternative Definition von λ-Termen.
`NFTerms.v`	Alternative Definition von λ-Termen in β-Normalform.
`Types.v`	Definition der einfachen Typen.
`Typing.v`	Definition der Typableitung für einfache Typen, Terme und NFTerme.
`LongTyping.v`	Definition der Langableitung.
`Filtr.v`	Teilformelfiltration und Implementierung von Lemma 3.18.
`Paths.v`	Definition von und Aussagen über Pfade in Termen.
`SfC.v`	Verifikation des Teilformelkalküls bis einschließlich Lemma 3.32, sowie Stubs zu den folgenden Lemmata.

Tabelle 4.1: Übersicht der Quelltextdateien

nicht angegeben, sondern in den wichtigsten Fällen beschrieben. Lediglich der Typ der Lemmata, also die Aussage der Lemmata, wird hier abgedruckt. Konstruktionen und Funktionen werden jedoch explizit angegeben und diskutiert.

4.2 `Type`, `Prop` und `Set`

Coq ist eine Sprache mit *higher order dependent typing*. Wir werden auf die genaue Theorie hinter Coq nicht im Detail eingehen, jedoch flossen einige Eigenschaften Coqs in Designentscheidungen der Formalisierung ein, die hier kurz erörtert werden.

Der Begriff *higher order dependent typing* bedeutet im Kern, dass wir sowohl Typen als Objekte betrachten können, die ihrerseits wieder Typen haben können (*higher order*), sowie, dass wir Objekte als Typen nutzen können, sodass ein Typ abhängig von dem Wert eines Objektes sein kann (*dependent typing*). Die Zahl 2 ist somit ein gültiger Typ und der Datentyp `nat` der natürlichen Zahlen ist ein gültiges Objekt, das einen eigenen Typen hat.

Der allgemeinste Typ, den Coq kennt, ist `Type`. Er agiert als Metatyp für Datentypen, die in Coq aufgestellt und genutzt werden. Weitere allgemeine Typen, die in Coq als Metatypen genutzt werden, sind `Prop` und `Set`. Da `Type` der allgemeinste Typ ist,

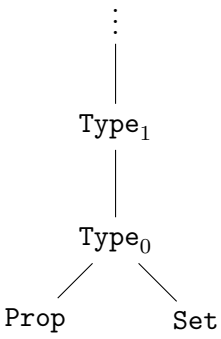


Abbildung 4.1: Hierarchie von `Type`, `Prop` und `Set`

gilt `Prop` $\prec$ `Type` und `Set` $\prec$ `Type`, und `Prop`, `Set` und `Type` haben jeweils den Typen `Type`.

```
IN: Check Prop.

OUT: Prop : Type

IN: Check Set.

OUT: Set : Type

IN: Check Type.

OUT: Type : Type
```

Dies führt dazu, dass `Type` sich selbst typt und über *Girards Paradox* dazu, dass das Typsystem von Coq inkonsistent wäre[10]. Dieses Problem wird in Coq dadurch umgangen, dass es eine Hierarchie an `Type`-Typen gibt. Jeder Typ `Type` der Ebene n wird von einem Typ `Type` der Ebene $n+1$ getypt. Des Weiteren gilt $\texttt{Type}_n \prec \texttt{Type}_{n+1}$. Siehe hierzu Abbildung 4.1. In Coq wird diese Hierarchie Universum genannt und üblicherweise vor dem Anwender versteckt.

Für Datentypen wird in Coq üblicherweise der Typ `Set` verwendet. So hat Beispielsweise der Typ der natürlichen Zahlen `nat` den Typen `Set`.

```
IN: Check nat.

OUT: nat : Set
```

Über den *Curry-Howard-Isomorphismus* wissen wir, dass wir Aussagen als Typen auffassen können und Beweise als λ-Terme, die diesen Typen inhabitieren[3]. Das Typsystem von Coq ist deutlich mächtiger als der einfach getypte Lambda-Kalkül und entspricht somit auch einer deutlich ausdrucksstärkeren Logik als dem implikativen Fragment der intuitionistischen Logik. Für Aussagen, die in Coq formalisiert werden, wird üblicherweise der Typ `Prop` verwendet.

```
IN: Check 3 < 7.

OUT: 3 < 7 : Prop
```

Während `Type`, `Prop` und `Set` als allgemeine Typen ähnlich verwendet werden können, ergeben sich Probleme, wenn in sie in Konstruktionen gemischt werden. Der Unterschied in der Behandlung von `Set` und `Prop` liegt darin, dass Coq für Typen in `Set` Inhabitanten voneinander unterscheidet, für Typen in `Prop` jedoch nicht. Es ist klar, dass wir zwischen zwei natürlichen Zahlen unterscheiden wollen, aber für Aussagen ist es für uns üblicherweise nur von Interesse, ob sie gelten, oder nicht, also ob ihr Typ inhabitiert wird, oder nicht. Diese Art von Beweisirrelevanz wird genutzt, wenn aus Coq verifizierte Programme exportiert werden.

Wenn wir nun die Existenz eines Inhabitanten für einen Typen in `Prop` beweisen wollen, können wir dafür Fallunterscheidungen über Beweise zu Aussagen in `Prop` treffen. Da der äußere Beweis nicht mitexportiert wird, muss auch der innere Beweis nicht exportiert werden. Wenn wir jedoch einen Inhabitanten eines Typen in `Set` aufstellen wollen, können wir dafür Inhabitanten von Typen in `Prop` nicht auseinandernehmen, da im Export dieser Teil des Inhabitantens fehlen würde.

Ein Problem ergibt sich insbesondere aus existentiell quantifizierte Aussagen sowie Disjunktionen. Eine Aussage der Form $\exists x.\varphi$, sowie $\psi \vee \varphi$ kann mehrere verschiedene Beweise haben. Der korrespondierende Typ kann somit mehrere verschiedene Inhabitanten haben. In Coq ist der resultierende Typ beider Konstruktionen in `Prop`. Somit kann für Beweise in `Set` die Konstruktion nicht auseinandergenommen werden. Für Konstruktionen in `Set` werden stattdessen `sumbool` und `sigT` verwendet. Anstelle `A ∨ B : Prop` kann `{A} + {B} : Set` und anstelle `∃ x, A. P : Prop` kann `{x : A & P} : Type` verwendet werden.

Betrachten wir die Definition der relevanten Typen in Definition 3.10. Das Enthaltensein in der Menge $T(\mathcal{D})$ wird über eine Eigenschaft aus $\mathcal{D}$ bestimmt. Da wir

[3] Siehe Abschnitt 2.3.1.

die verschiedene Elemente der Menge unterscheiden wollen, eignet sich für $T(\mathcal{D})$ eine Konstruktion in `Set`. Dadurch können wir die Typableitung, über die $\mathcal{D}$ definiert ist, nicht in `Prop` definieren. Aus diesem Grund wählen wir `Type` als Metatyp für unsere Konstruktionen und damit auch für unsere Aussagen, da `Type` sowohl ein Supertyp von `Set` als auch ein Supertyp von `Prop` ist. Dies erlaubt es uns, an einigen Stellen Lemmata der Standardbibliothek zu nutzen, die für `Prop` definiert sind, aber gleichzeitig in Beweisen und Konstruktionen $\mathcal{D}$ auseinanderzunehmen. Zwar können so viele Lemmata, die Aussagen über Typen in `Prop` treffen weiterverwendet werden; Konstruktionen, die für `Prop` definiert wurden und in eigenen Typen verwendet werden, mussten jedoch in `Type` neu implementiert werden. Diese Neuimplementierungen, sowie entsprechende Lemmata zu diesen, finden sich zum Großteil in der Datei `Utils.v`.

4.3 Datentypen

Wir betrachten zunächst die grundlegenden Datentypen, die wir zur Formalisierung des einfach getypten Lambda-Kalküls benötigen, also λ-Terme und einfache Typen, wobei für erstere verschiedene Definitionen aufgestellt werden. Anschließend werden Typ- und die Langableitung auf den jeweiligen Termmodellen definiert.

4.3.1 λ-Terme

Anstelle der λ-Terme, wie sie in Definitionen 2.1 und 2.11 definiert sind, nutzen wir Terme in *De-Bruijn-Notation*, gemäß Definition 2.49.

Listing 4.1: `Terms.v` **Siehe Definition 2.49**

```
16 Inductive term :=
17   | Var (x : var)
18   | App (p q : term)
19   | Lam (s : {bind term}).
```

Bemerkung: Der Datentyp `var` ist nur ein anderer Name für `nat`.

Betrachten wir den Konstruktor `Lam` in Listing 4.1, so stellen wir fest, dass der Parameter nicht nur vom Typ `term` ist, sondern vom Typ `bind term`. Dies annotiert den inneren Term mit seiner Bindungstiefe und wird von Autosubst intern benutzt. Wir können dies an dieser Stelle ignorieren und `s` wie einen regulären Term behandeln.

Anstelle die Konstruktoren des Datentyps direkt zu nutzen, nutzen wir Coqs Möglichkeit, kompaktere Notationen zu definieren.

Listing 4.2: Terms.v

```
22 Notation "'!' x" := (Var x) (at level 15).
23 Notation "p '@' q" := (App p q)
24     (at level 31, left associativity).
25 Notation "'\_' p" := (Lam p) (at level 35, right associativity).
```

Die Terme **I**, **K** und **S** lassen sich nun folgendermaßen in Listing 4.3 darstellen.

Listing 4.3: Terms.v **Siehe Beispiel 2.50**

```
31 Definition tI := \_ !0.
32 Definition tK := \_ \_ !0.
33 Definition tS := \_\_\_((!2@!0)@(!1@!0)).
```

Autosubst definiert Klassen, über denen Substitutionen definiert sind. Mittels der Taktik `derive` können, sofern der entsprechende Datentyp nicht zu komplex ist und mittels `var` und `bind` definiert ist, automatisch diese Klassen erfüllt werden. Die Instanziierung der Klassen ist in Listing 4.4 zu sehen.

Listing 4.4: Terms.v

```
26 Instance Ids_term : Ids term. derive. Defined.
27 Instance Rename_term : Rename term. derive. Defined.
28 Instance Subst_term : Subst term. derive. Defined.
29 Instance SubstLemmas_term : SubstLemmas term. derive. Qed.
```

Über die Klasse `Subst` ist die Substitutionsfunktion `subst` definiert, die eine Variablensubstitutionsfunktion auf Terme fortsetzt.

```
IN: Check subst.

OUT: subst : (var → term) → term → term
```

Dies entspricht unserer Substitution gemäß Definition 2.8. Auf dieser Basis kann nun in Listing 4.5 die β-Reduktion implementiert werden.

Listing 4.5: `Terms.v` **Siehe Definition 2.14**

```
73 Inductive step : term → term → Prop :=
74   | Step_beta (s1 s2 t : term) :
75       s1.[t/] = s2 → step (App (Lam s1) t) s2
76   | Step_appL (s1 s2 t : term) :
77       step s1 s2 → step (App s1 t) (App s2 t)
78   | Step_appR (s t1 t2 : term) :
79       step t1 t2 → step (App s t1) (App s t2)
80   | Step_lam (s1 s2 : term) :
81       step s1 s2 → step (Lam s1) (Lam s2).
```

Bemerkung: `s1.[t/]` entspricht `subst (t .: ids) s1`. Die Funktion `ids : var → term` ist über die Klasse `Ids` definiert und entspricht einer Substitution, die einer Variablen `n` ihren den Term `! n` zuweist. Wird diese auf Terme geliftet, entspricht dies der Identität. Der Operator `.:` entspricht der Funktion `scons (m : term) (Su : var → term) : var → term` und verschiebt die Zuweisungen einer übergebenden Funktion `Su` um 1 und überschreibt die erste Position mit `m`.
Eine Substitution wird im Grunde über die Funktion aus den natürlichen Zahlen als unendliche Liste interpretiert, die für jede Variable – also natürliche Zahl – einen Wert hat.
Da in der Reduktion eine Abstraktion entfernt wird, sinkt der Abstand aller inneren Variablen zu ihrer entsprechenden Abstraktion um 1. Dies wird mit dem Verschieben um 1 gelöst und die ehemalige Variable 0 entspricht genau der Variable, die durch die Reduktion ersetzt wird.

Über die Reduktion lässt sich nun der Begriff der β-Normalform in Listing 4.6 definieren.

Listing 4.6: `Terms.v`

```
143 Definition NF (t : term) := ∀ t', ¬step t t'.
```

Terme mit zwei Konstruktoren

Betrachten wir Definitionen 2.40 und 3.21, so stellen wir fest, dass die Ableitungsregeln im Gegensatz zu denen aus Definition 2.28 auf Basis von zwei, anstelle von drei Konstruktoren definiert sind.

Eine Möglichkeit, λ-Terme auf Basis von zwei Konstruktoren zu definieren, ist es, die Terme grundsätzlich als Liste von Applikationen zu sehen. Wir haben nun entweder eine Variable oder eine Abstraktion, auf die jeweils eine Liste von Termen angewendet wird. Listing 4.7 zeigt die Implementierung solcher Terme.

Listing 4.7: `Terms2.v`

```
9  Inductive term2 :=
10   | App_var (ms : list term2) (x : var)
11   | App_lam (ms : list term2) (s : {bind term2}).
```

Bemerkung: Die Konstruktoren erwarten zunächst eine Liste von Termen und anschließend eine Variable oder eine Abstraktion. Dies vereinfacht AUTOSUBST die Generierung von Substitutionslemmata.

Auch hier können wir eine Notation einführen. Dies ist in Listing 4.8 zu sehen.

Listing 4.8: `Term2.v`

```
37 Notation "'!!!!' x" := (App_var [] x) (at level 15).
38 Notation "'!!!' p '@@' q" := (App_var q p)
39   (at level 31, left associativity).
40 Notation "'\___' p" := (App_lam [] p)
41   (at level 35, right associativity).
42 Notation "'\__' p '@@' q" := (App_lam q p)
43   (at level 35, right associativity).
```

Die Datei `Terms2.v` enthält die Implementierung für diese Art von Terme, inklusive der folgenden Funktionen, die beide λ-Termvarianten ineinander überführen, sowie Lemmata, die zeigen, dass beide Definitionen isomorph zueinander sind und einigen weiteren Lemmata und Konstruktionen, die es erlauben, Substitutionen auf

den Termen durchzuführen. Diese sind in Listing 4.9 gelistet, hierbei ist >>> die Funktionskomposition.

Listing 4.9: `Terms2.v`

```
43 Fixpoint term2_term (m : term2) : term

49 Fixpoint term_term2 (m : term) : term2

60 Lemma term2_term_id : ∀ m, (term2_term >>> term_term2) m = m.

88 Lemma term_term2_id : ∀ m, (term_term2 >>> term2_term) m = m.

101 Lemma term2_term_if : ∀ m n,
102     term2_term m = term2_term n → m = n.

111 Lemma term_term2_if : ∀ m n,
112     term_term2 m = term_term2 n → m = n.
```

Die Konstruktion aus `Term2.v` erlaubt es uns für die (long) und $(\text{varapp})_R$ Regel die Konstruktion `!! x @@ [m_1;...;m_n]` und für die (abs) und $(\text{abs})_R$ die Notation `\___ m` zu verwenden. Im Fall der Abstraktion können wir jedoch beobachten, dass sowohl für die Lang-, als auch für die Pfadableitung, keine Terme der Form `\__ s @@ [m_1, ... ,m_n]` benutzt werden. Die in `Term2.v` definierten Terme haben zwar genau zwei Konstruktoren und sind isomorph zu den in `Term.v` definierten Termen, passen aber immer noch nicht genau auf die Terme, die in der Langableitung und dem Teilformelkalkül erwartet werden. Aus diesem Grund werden wir diese Terme in unserer weiteren Implementierung nicht nutzen. Sie stellen jedoch einen wichtigen Schritt für die folgende Implementierung von λ-Termen dar.

Terme in Normalform

Wenn wir nun verbieten, dass Abstraktionen als erstes Element einer Applikation vorkommen, erhalten wir eine Art von λ-Termen, deren Konstruktoren genau auf die Lang- und Pfadableitung passen, die in Listing 4.10 präsentiert wird.

Listing 4.10: NFTerms.v

```
11 Inductive nfterm :=
12   | NFcurr (ms: list nfterm) (x : var)
13   | NFLam (s: {bind nfterm}).
```

Es fällt auf, dass mit dieser Definition der λ-Terme, nicht alle Terme abgebildet werden können. Grundsätzlich können wir keine Terme darstellen, die Redexe haben (siehe Definition 2.13). Daraus folgt, dass wir nur Terme darstellen können die in β-Normalform sind. Dies stellt sich jedoch als kein Problem heraus, da wir uns weitestgehend auf Terme in η-langer Normalform beschränken. Die in `NFTerms.v` definierten Funktionen `NFterm_term (nft : nfterm) : term` und `term_NFterm (t : term) : option term` bilden von den regulären λ-Termen in die NFTerme ab. Dies erlaubt es uns Resultate über den Termen, die in diesem Abschnitt definiert werden, auf die regulären λ-Terme zu übertragen.

Da die Terme aus `Terms2.v` (siehe Listing 4.7) und `NFTerms.v` (siehe Listing 4.10) niemals gleichzeitig genutzt werden, kann die Notation aus `Terms2.v` auch für `nfterm` wiederverwendet werden.

Listing 4.11: NFTerms.v

```
59 Notation "'!!' x '@@' ms" := (NFcurr ms x)
60   (at level 31, left associativity).
61 Notation "'\__' s" := (NFLam s)
62   (at level 35, right associativity).
```

Coq generiert für Typen automatisch Induktionsprinzipien, die von der Taktik `induction` genutzt werden. Das Induktionsprinzip, das Coq für `nfterm` generiert, ist jedoch nicht stark genug, um Aussagen über die Terme in der Liste der Applikation zu beweisen.

```
In: Check nfterm_ind.
Out:

nfterm_ind : ∀ P : nfterm → Prop,
  (∀ (ms : list nfterm) (x : var), P (!! x @@ ms)) →
```

```
    (∀ s : {bind nfterm}, P s → P (\__ s)) →
      ∀ n : nfterm, P n
```

Dem generierten Induktionsprinzip fehlt die Voraussetzung, dass die Bedingung `P` auch für jeden Term `m` aus `ms` gelten muss. Aus diesem Grund nutzen wir ein eigenes Induktionsprinzip `nfterm_ind'` (Listing 4.12), das dieses fordert.

Listing 4.12: NFTerms.v

```
26 Definition nfterm_ind' : ∀ P : nfterm → Prop,
27     (∀ (ms : list nfterm) (x : var),
28       (Forall P ms) → P (!! x @@ ms)) →
29         (∀ s : {bind nfterm}, P s → P (\__ s)) →
30           ∀ n : nfterm, P n
```

Da NFTerme immer in β-Normalform sind, wird auch keine Reduktionsrelation wie `step` (Listing 4.5) über `nfterm` definiert. Der Datentyp `nfterm` wird auch nicht als Instanz der Substitutionsklassen von Autosubst formuliert. Syntaktisch ist dies auch nicht möglich, da durch Substitution aus einem Term in β-Normalform ein Term entstehen kann, der nicht in β-Normalform ist. Dieser lässt sich nicht mit dem Datentyp `nfterm` darstellen. Da für das Untersuchen der prinzipalen Inhabitation innerhalb von Termen nicht substituiert werden muss, ist dies ebenfalls kein Problem.

Lemma 3.26 zeigt, dass wir nur geschlossene Terme betrachten müssen. Definition 2.5 definiert geschlossene Terme dadurch, dass die Menge der freien Variablen leer sein muss. Wir benötigen also eine Formalisierung freier Variablen. Hierfür können wir die Eigenschaft der De-Bruijn-Notation ausnutzen, dass gebundene Variablen einfach durchnummeriert sind. Wir zählen die Anzahl der Abstraktionen bis zu einer Variable und ziehen diese vom Wert der Variable ab. Wenn hier ein Wert herauskommt, der echt größer als 0 ist, ist die Variable an keine Abstraktion gebunden und ist frei. In unserer Behandlung treten freie Variablen genau dann auf, wenn wir in einen Abstraktionsableitungsschritt durchführen. Hierdurch werden für die freien Variablen fortlaufende Nummern verwendet. Anstelle die Menge der freien Variablen zu errechnen, reicht es, dass wir die größte freie Variable errechnen. Ist diese 0, wissen wir, dass es sich um einen geschlossen Term handelt. Dies ist in Listing 4.13 implementiert.

Listing 4.13: NFterms.v — **Sieht Definition 2.5**

```
224 Fixpoint max_fvar (m: nfterm) : var :=
225   match m with
226   | !! x @@ ms ⇒ fold_left Nat.max (map max_fvar ms) (S x)
227   | \__ s ⇒ pred (max_fvar s)
228   end.

231 Definition closed m := max_fvar m = 0.
```

4.3.2 Typen

Auch für die einfachen Typen wird hier ein Datentyp definiert sowie eine vereinfachte Notation angeben. Dies geschieht in Listing 4.14 und 4.15. Analog zu den Termen wird als Variablenmenge die Menge der natürlichen Zahlen verwendet.

Listing 4.14: Types.v — **Siehe Definition 2.24**

```
13 Inductive type :=
14   | Atom (x: var)
15   | Arr (A B : type).
```

Listing 4.15: Types.v

```
22 Notation "'?' x" := (Atom x) (at level 15).
23 Notation "a '⇝' b" := (Arr a b)
24   (at level 51, right associativity).
```

Die kanonischen Typen der **S**-, **K**- und **I**-Kombinatoren können dann folgendermaßen dargestellt werden.

Listing 4.16:

```
1 Definition IType := ? 0 ~> ? 0.
2 Definition KType := ? 0 ~> ? 1 ~> ? 0.
```

```
3 Definition SType :=
4     (? 0 ~> ? 1 ~> ? 2) ~> (? 0 ~> ? 1) ~> ? 0 ~> ? 2.
```

Bemerkung: Im Gegensatz zu den Termen in De-Bruijn-Notation ist die Nummerierung der atomaren Variablen nicht relevant. Der Typ `? 0 ~> ? 0` entspricht genau dem Typen `? 1 ~> ? 1`. Im Gegensatz zu λ-Termen ist das Behandeln von Typen, die bis auf Umbenennung gleich sind, deutlich einfacher, da die atomaren Variablen an keine Abstraktionen gebunden werden. Aus diesem Grund müssen Typen nicht unterscheiden werden, wenn sie bis auf Umbenennung gleich sind.

Des Weiteren benötigen wir eine Formalisierung der Pfade in Typen aus Definition 3.19, sowie den dort definierten Funktionen P_ρ und $\mathrm{dom}(P_\rho)$.

Listing 4.17: `Paths.v` **Siehe Definition 3.19**

```
14 Inductive dir :Type :=
15   | Src
16   | Tgt.
```

Pfade können nun als Listen von *Directions* dargestellt werden.

Listing 4.18: `Paths.v` **Siehe Definition 3.19**

```
28 Definition path : Type := list dir.
```

Für die Definition der Funktion P_ρ muss beachtet werden, dass dies eine partielle Funktion ist. Somit gibt es Pfade, deren Wert undefiniert ist. Dies wird mit dem Datentyp `option` in Listing 4.19 gelöst.

Listing 4.19: `Paths.v` **Siehe Definition 3.19**

```
32 Fixpoint P (rho:type) (pi: path) {struct pi} : option type :=
33   match pi with
34   | [] ⇒ Some rho
```

```
35  | Src::pi' ⇒ match rho with
36                | (? x) ⇒ None
37                | sigma ~> _ ⇒ P sigma pi'
38                end
39  | Tgt::pi' ⇒ match rho with
40                | (? x) ⇒ None
41                | _ ~> tau ⇒ P tau pi'
42                end
43  end.
```

Um festzustellen, für welche Pfade ein Typ definiert ist, bedarf es einer Implementierung der Funktion $\mathrm{dom}(P_\rho)$.

Listing 4.20: `Paths.v` **Siehe Definition 3.19**

```
58 Fixpoint dom_P (rho: type) : list path :=
59   match rho with
60   | ? x ⇒ [[]]
61   | sigma ~> tau ⇒ [] :: map (cons Src) (dom_P sigma) ++
62       map (cons Tgt) (dom_P tau)
63   end.
```

Es kann nun bewiesen werden, dass `dom_P` (Listing 4.20) genau dem Definitionsbereich der Funktion `P` (Listing 4.19) entspricht.

Listing 4.21: `Paths.v`

```
64 Lemma dom_P_some : ∀ pi rho, In pi (dom_P rho) →
65     { rho' & P rho pi = Some rho'}.

80 Lemma dom_P_none : ∀ pi rho, ¬ In pi (dom_P rho) → P rho pi =
   ↪  None.
```

Auf Basis der in Listing 4.21 aufgestellten Lemmata und `dom_P` kann nun eine Funktion definiert werden, die einen Beweis entgegennimmt, der zeigt, dass der

übergebende Pfad in dem Definitionsbereich von P_ρ liegt, und den entsprechenden Teiltypen zurückgibt.

Listing 4.22: Paths.v

```
159 Definition P_ok rho pi (proof : In pi (dom_P rho)) : type.
```

Anschließend muss noch gezeigt werden, dass sich `P_ok` und `P` gleich verhalten. Über das in Lemma aus Listing 4.23 kann insbesondere ein Beweis für `P_ok` in einen Beweis für `P` überführt werden, da für einige Situationen die Konstruktion als partielle Funktion und für andere Situationen die Konstruktion mit explizitem Beweis besser geeignet ist.

Listing 4.23: Paths.v

```
194 Lemma P_ok_P {rho pi rho' pr}: P_ok rho pi pr = rho' ↔ P rho pi
    ↪  = Some rho'.
```

Es ist klar, dass der genaue Beweis für das Enthaltensein eines Pfades in `dom_P` nicht wichtig für `P_ok` ist. Diese Intuition beweist das folgende Lemma in Listing 4.24. Es erlaubt insbesondere größere Flexibilität bei der Konstruktion der Beweise.

Listing 4.24: Paths.v

```
181 Lemma P_ok_proof_irl : ∀ rho pi p1 p2,
182     P_ok rho pi p1 = P_ok rho pi p2.
```

Zuletzt betrachten wir eine Implementierung von Teiltypen, ohne explizit einen Pfad anzugeben.

Listing 4.25: Types.v

```
125 Inductive subformula : type → type → Prop :=
126   | subf_refl : ∀ rho, subformula rho rho
127   | subf_arrow_l : ∀ rho sigma tau,
128       subformula rho sigma → subformula rho (sigma ~> tau)
```

```
129   | subf_arrow_r : ∀ rho sigma tau,
130       subformula rho tau → subformula rho (sigma ~> tau).
```

Für die spätere Handhabung ist es wichtig, dass entschieden werden kann, ob ein gegebener Typ Teiltyp eines anderen Typen ist.

Listing 4.26: `Types.v`

```
131 Theorem subformula_dec : ∀ x y,
132     {subformula x y} + { ¬ subformula x y}.
```

4.3.3 Ableitungen

Wir betrachten hier die Typableitungen aus Definitionen 2.28 und 2.40, sowie die Ableitung des Teilformelkalküls (Definition 3.21). Da wir verschiedene Implementierungen für Terme haben, benötigen wir für jede eine eigene Typisierung. Die Teilformelfiltration ist über den Datentyp `term` (Listing 4.1) definiert. Aus diesem Grund benötigt der Typ `term` die reguläre Typableitung. `term` wird auch genutzt, um einen Zusammenhang zwischen langen Inhabitanten und regulären Inhabitanten zu zeigen, siehe hierfür Lemmata 2.41 und 2.42. Somit muss auch eine Langableitung für `term` formuliert werden. Der Teilformelkalkül wird ausschließlich über `nfterm`e (Listing 4.10) definiert. Wir benötigen somit noch eine Langableitung und die Pfadableitung für `nfterm`.

Typableitung

Zunächst benötigen wir eine Formalisierung für Typkontexte. In Definition 2.27 haben wir einen Typkontext als Relation über die Variablenmenge und den einfachen Typen definiert, sodass jeder Variable maximal ein Typ zugewiesen wird. Ein Ansatz zur Formalisierung wäre es, den Typkontext als partielle Funktion `Gamma : var → option type` zu implementieren. Betrachten wir hier aber den Definitionsbereich `var`, bzw. `nat`, stellen wir fest, dass dies dem Nachschlagen in einer Liste von Typen entspricht. `nth_error Gamma : nat → option type` für `Gamma : list type`. Über das Generierungslemma 2.29 wissen wir, dass in einer Typableitung ein Typkontext immer nur mit der Variable der äußersten Abstraktion befüllt wird. In De-Bruijn-Notation entspricht dies der Variable 0, falls die Variable nicht in einer weiteren Abstraktion

referenziert wird. Ansonsten können wir den Kontext entsprechend verschieben. Wir können also Typkontexte als Liste von Typen auffassen.

Listing 4.27: `Types.v` **Siehe Definition 2.27**

```
25 Definition repo := list type.
```

Auf dieser Basis kann nun eine Formalisierung für die Typableitung formuliert werden.

Listing 4.28: `Typing.v` **Siehe Definition 2.28**

```
18 Inductive ty_T (Gamma : repo) : term → type → Type:=
19   | Ty_Var x rho : nth_error Gamma x = Some rho →
20                 ty_T Gamma (! x) rho
21   | Ty_Lam s sigma tau : ty_T (sigma :: Gamma) s tau →
22                 ty_T Gamma (\_ s) (sigma ~> tau)
23   | Ty_App s t sigma tau : ty_T Gamma s (sigma ~> tau) → ty_T
     ↪  Gamma t sigma →
24                 ty_T Gamma (s @ t) tau.
```

Das folgende Beispiel zeigt, wie sich über die Darstellung des Kontextes als Liste und der Terme in De-Bruijn-Notation ein Kontext über Abstraktionen aufbaut.

Beispiel 4.29

Betrachten wir den Term `\_ \_ ! 1` und den Typen `? 0 ~> ? 1 ~> ? 0`. Es gilt `ty_T [] (\_ \_ ! 1) (? 0 ~> ? 1 ~> ? 0)`, da `ty_T [? 0] (\_ !1) (? 1 ~> ? 0)`, `ty_T [? 1; ? 0] (! 1) (? 0)` und `nth_error [? 1; ?0] 1 = Some (? 0)`.

Bemerkung: Listing 4.28 definiert einen Datentypen für die Typableitung, somit entspricht dieser Datentyp genau der Typableitung $\mathcal{D}$ aus Definition 2.28.

Da über Autosubst eine Substitutionsfunktion für Typen definiert ist, kann nun der Begriff des prinzipalen Typs in Listing 4.30 formalisiert werden.

Listing 4.30: `Typing.v` **Siehe Definition 3.3**

```
36 Definition princ rho m: Type :=
37     ty_T [] m rho * (∀ rho', ty_T [] m rho' → {Su & rho.[Su] =
     ↪  rho'}).
```

Bemerkung: Das Produkt zweier Typen `A : Type` und `B : Type`, `A * B : Type` entspricht der Konjunktion der Aussagen. Hier wird das Produkt anstelle der direkten Konjunktion `A / B` gewählt, um explizit einen Datentypen in `Type` zu erhalten.

Für `ty_T` (Listing 4.28) können wir nun auch Generierungslemmata aufstellen.

Listing 4.31: `Typing.v` **Siehe Lemma 2.29**

```
40 Lemma generation_app_T : ∀ s t tau (Gamma : repo), ty_T Gamma
   ↪  (s@t) tau →
41     {sigma & (ty_T Gamma s (sigma ~> tau)) * (ty_T Gamma t
     ↪  (sigma)) }.

50 Lemma generation_lam_T : ∀ s rho (Gamma : repo) sigma tau,
51     ty_T Gamma (\_ s) rho → rho = sigma ~> tau → ty_T (sigma ::
     ↪  Gamma) s tau.

58 Lemma generation_var_T : ∀ x rho (Gamma : repo), ty_T Gamma (!
   ↪  x) rho →
59     nth_error Gamma x = Some rho.
```

Typableitung für NFTerme

Bevor wir die Implementierung für Typableitungen der `nfterm` (Listing 4.10)e betrachten, stellen wir die theoretische Grundlage dieser auf.

Lemma 4.32: Typableitung für Terme in β-Normalform

Sei $\Gamma \subseteq \mathcal{V} \times \mathbb{T}$ ein Typkontext, $\sigma_1, \ldots, \sigma_n, \tau \in \mathbb{T}$ einfache Typen, $M, M_1, \ldots M_n \in \Lambda$ λ-Terme und $x \in \mathcal{V}$ eine Variable. Die folgenden Ableitungsregeln typen genau λ-Terme in β-Normalform.

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau} \text{ (abs)}$$

$$\frac{\Gamma, x : \sigma_1 \to \ldots \to \sigma_n \to \tau \vdash M_i : \sigma_i \text{ für } i = 1 \ldots n}{\Gamma, x : \sigma_1 \to \ldots \to \sigma_n \to \tau \vdash x\ M_1\ \ldots\ M_n : \tau} \text{ (varapp)}$$

Beweisidee: Die Ableitungsregeln ähneln stark denen aus Definition 2.40. Sie typen – genau wie die lange Typbarkeit – genau Terme in β-Normalform, wir erlauben jedoch Zieltypen, die keine atomaren Typen sind. Analog zu dem Beweis zu Lemma 2.41 lässt sich zeigen, dass diese Typisierung der Typisierung aus Definition 2.28 entspricht.

Während die mehrfache Applikation in Termen direkt im Konstruktor von `nfterm` über eine Liste implementiert wird, benötigen wir noch eine Konstruktion, die entsprechende Typen beschreibt. Dies wird hier mittels einer Funktion `make_arrow_type` gelöst, der eine Liste von Typen übergeben wird, um den Typen $\sigma_1 \to \ldots \to \sigma_n \to \tau$ zu erstellen.

Listing 4.33: `Types.v`

```
222 Definition make_arrow_type (ts : list type) (a : type) :=
223     fold_right Arr a ts.
```

Nun kann die Typisierung der `nfterm`e angeben werden. Diese wird zwar nicht in der restlichen Implementierung verwendet, ist aber ein wichtiger Schritt auf dem Weg, die Langableitung für `nfterm` zu formulieren.

Listing 4.34: `Typing.v`

```
27 Inductive nfty (Gamma : repo) : nfterm → type → Type :=
28   | NFTy_lam s sigma tau :
29       nfty (sigma :: Gamma) s tau → nfty Gamma (\__ s) (sigma
       ↪  ~> tau)
```

```
30  | NFTy_var x tau ts ms :
31      nth_error Gamma x = Some (make_arrow_type ts tau) →
32      length ms = length ts →
33      (∀ n (pms : n < length ms) (pts : n < length ts),
34         nfty Gamma (nth_ok ms n pms) (nth_ok ts n pts)) →
35      nfty Gamma (!!x @@ ms) tau.
```

Bemerkung:

- Die Funktion `nth_ok` ist eine Variante von `nth_error`, die analog zu `P` (Listing 4.19) und `P_ok` (Listing 4.22) einen Beweis erwartet, der aussagt, dass der übergebene Wert im Definitionsbereich der partiellen Funktion liegt, also dass `proof : n < length ms` gilt. Die Datei `Utils.v` enthält die Definition sowie diverse Aussagen zu `nth_ok`.
- Da wir im `NFTy_var`-Fall fordern, dass die Liste der Terme genauso lang ist wie die Liste der Typen, ist es redundant zu fordern, dass `n` sowohl kleiner als `length ms` und `length ts` ist. Ein Beweis für die eine Aussage lässt sich aus dem Beweis für die jeweils andere Aussage generieren. Die hier gewählte Variante erfordert, dies bei jedem Aufruf zu tun, vereinfacht aber die Definition von `nfty`. Da `nfty` in der restlichen Arbeit wenig genutzt wird, wird hier eine klarere Definition statt der einfachen Handhabung bevorzugt. Wir werden in der Definition der Langableitung für `nfterm` eine andere Variante kennenlernen.

Der Ableitungsschritt für die Abstraktion hier gleicht dem Ableitungsschritt für die Abstraktion von `ty_T` (Listing 4.28). Der Ableitungsschritt für die Applikation ist jedoch komplizierter. Wir werden ihn daher genauer betrachten. Die Regel (varapp) hat eine von n abhängige Anzahl an Hypothesen, sie kann somit nicht als einfache Folgerung notiert werden. Wir können jedoch feststellen, dass in allen Hypothesen gefordert wird, dass x in Γ mit $\sigma_1 \to \ldots \to \sigma_n \to \tau$ getypt wird. Dies kann mittels `nth_error Gamma x = Some (make_arrow_type ts tau)` als eigenständige Bedingung formuliert werden. Grundsätzlich muss noch gefordert werden, dass die Anzahl der Eingabetypen für x, sowie die Anzahl der Terme, auf die x appliziert wird, jeweils gleich n sind. Dies entspricht der Forderung, dass `length ms = length ts` gilt. Es bleibt nun die variable Anzahl an Hypothesen von (varapp) zu formalisieren. Es lässt sich feststellen,

dass alle Hypothesen dieselbe Struktur haben und sich nur in dem Parameter i unterscheiden. Dies führt zu der Bedingung `(∀ n (pms : n < length ms) (pts : n < length ts), nfty Gamma (nth_ok ms n pms) (nth_ok ts n pts))`[4] Diese Art der Formulierung von Ableitungsregeln mit variablen Hypothesen werden für die Implementierung der Langableitung und des Teilformelkalküls ebenfalls verwenden werden.

Über Lemma 2.36 haben wir eine weitere Charakterisierung von geschlossenen Termen gefunden. Insbesondere gibt der Beweis einen Zusammenhang zwischen der Anzahl der freien Variablen und dem Typkontext. Da der Typkontext als Liste von Typen implementiert ist, entspricht die Stelle im Typkontext der entsprechenden Variable. Freie Variablen werden in dieser Implementierung kontinuierlich vergeben, somit können wir über den Vergleich von `max_fvar` (Listing 4.13) und der Größe des `repos` (Listing 4.27) feststellen, ob ein Typkontext alle freien Variablen typt.

Listing 4.35: `NFTerms.v`

```
230 Definition all_var_in_repo {A} m (Delta : list A) :=
231     max_fvar m < S (length Delta).
```

Bemerkung:

- `Delta` ist nicht zwangsläufig eine Liste von Typen, sondern kann ein beliebiger Kontext sein.
- Für `closed` (Listing 4.13) ist `Delta = []`, damit muss `max_fvar m < 1`, also 0 sein.
- Diese Darstellung vereinfacht es Induktionsbeweise in Coq zu führen, da Coq für Induktionsbeweise erfordert, dass das Konstrukt, über den eine Induktion durchgeführt möglichst allgemein ist. Da für `closed` gefordert wird, dass `Delta = []`, kann mit `all_var_in_repo` eine Eigenschaft für allgemeinere Konstrukte beschrieben werden.

Durch diese Darstellung können einfacher Induktionsbeweise geführt werden, deren Aussage über geschlossene Term formuliert ist.

[4] In der Implementierung wird anstelle der Variable i die Variable `n` verwendet.

Langableitung für λ-Terme

Die Langableitung ist nur auf einer Einschränkung der λ-Terme definiert. Für die Implementierung mittels `term` (Listing 4.1) benötigen wir somit – analog zu der Funktion `make_arrow_type` (Listing 4.33) – eine Funktion, die einen Term auf eine Liste von Termen anwendet.

Listing 4.36: `Terms.v`

```
226 Definition curry (x:term) (terms: list term) : term :=
227     fold_left App terms x.
```

Bemerkung: Es handelt sich hier tatsächlich um eine Art des *currying*. Ein Term x, der auf eine Liste von Argumenten angewendet wird, wird interpretiert als Term, auf den die Argumente stückweise angewendet werden.

Listing 4.37: `LongTyping.v` **Siehe Definition 2.40**

```
15 Inductive long_ty_T (Gamma : repo) : term → type → Type :=
16   | Long_I_T s sigma tau : long_ty_T (sigma :: Gamma) s tau →
17       long_ty_T Gamma (\_ s) (sigma ~> tau)
18   | Long_E_T x ms ts a :
19       nth_error Gamma x = Some (make_arrow_type ts (? a)) →
20         Forall2_T (long_ty_T Gamma) ms ts →
21           long_ty_T Gamma (curry (! x) ms) (? a).
```

Bemerkung:

- Im Gegensatz zu `nfty` (Listing 4.34) ist die Applikation nicht über $\forall$ und Beweisen über das Enthaltensein im Definitionsbereich von `nth_ok` definiert, sondern es wird mittels des Konstrukts `Forall2_T` gefordert, dass für die entsprechenden Paare aus `ms` und `ts` die Bedingung `long_ty_T Gamma` gilt. Beide Ansätze sind äquivalent.
- Das Konstrukt `Forall2_T` ist analog zu `Forall2` aus der Standardbibliothek definiert. Im Gegensatz zu Letzterem ist es aber in `Type` definiert, somit kann es in den Beweisen auseinandergenommen werden. Die De-

finition von `Forall2_T` sowie diverser Aussagen dazu befinden sich in `Utils.v`.

- Im Gegensatz zu Definition 2.40 wird hier nicht die Regel `Ty_Lam` (Listing 4.28) wiederverwendet, sondern neu angegeben, da wir mit der Langableitung einen vollständig neuen Typen erstellen.
- Die Menge Long(ρ) wird nicht explizit aufgestellt, da das Berechnen der gesamten Menge nicht erforderlich ist. Lediglich die Frage, ob ein gegebener Term in der Menge enthalten ist, muss beantwortet werden. Der Beweis, dass $M \in \text{Long}(\rho)$ gilt, entspricht einem Inhabitanten von `long_ty_T [] m rho`.

Genau wie bei der Definition des Datentyps `nfterm` (Listing 4.10) hat Coq für `long_ty_T` (Listing 4.37) ein zu schwaches Induktionsprinzip generiert.

```
In: Check long_ty_T_ind.
Out:

long_ty_T_ind :
  ∀ P :
    ∀ (Gamma : repo) (t : term) (t0 : type),
      long_ty_T Gamma t t0 → Prop,
    (∀ (Gamma : repo) (s : term) (A B : type)
        (l : long_ty_T (A :: Gamma) s B),
      P (A :: Gamma) s B l →
        P Gamma (\_ s) (A ~> B) (Long_I_T Gamma s A B l)) →
    (∀ (Gamma : repo) (x : nat) (ms : list term)
        (ts : list type) (a : var)
        (e : nth_error Gamma x =
             Some (make_arrow_type ts (? a)))
        (f0 : Forall2_T (long_ty_T Gamma) ms ts),
      P Gamma (curry (! x) ms) (? a)
        (Long_E_T Gamma x ms ts a e f0)) →
    ∀ (Gamma : repo) (t : term) (t0 : type)
```

```
        (l : long_ty_T Gamma t t0),
        P Gamma t t0 l
```

Wir haben im `Long_E_T`-Fall zwar die Voraussetzung `f0`, dass alle Subableitung korrekt getypt sind, aber nicht, dass `P` auch in den inneren Subableitungen gilt. Dem folgenden Induktionsprinzip in Listing 4.38 ist die Voraussetzung `Forall2_T (P Gamma) ms ts` hinzugefügt.

Listing 4.38: `LongTyping.v`

```
74 Definition long_ty_T_ind' :
75     ∀ P : repo → term → type → Type,
76       (∀ (Gamma : repo) (s : term) (A B : type),
77         long_ty_T (A :: Gamma) s B →
78           P (A :: Gamma) s B → P Gamma (\_ s) (A ~> B)) →
79       (∀ (Gamma : repo) (x : var)
80         (ms : list term) (ts : list type) (a : var),
81           nth_error Gamma x = Some (make_arrow_type ts (? a)) →
82             Forall2_T (long_ty_T Gamma) ms ts →
83               Forall2_T (P Gamma) ms ts →
84                 P Gamma (curry (! x) ms) (? a)) →
85       ∀ (Gamma : repo) (t : term) (t0 : type),
86         long_ty_T Gamma t t0 → P Gamma t t0
```

Bevor eine Formalisierung für Lemma 2.41 angegeben werden kann, muss ein Hilfslemma aufgestellt werden, das aussagt, dass die Ableitung `ty_T` (Listing 4.28) mit den Funktionen `curry` (Listing 4.36) und `make_arrow_type` (Listing 4.33) kompatibel ist.

Listing 4.39: `LongTyping.v`

```
117 Lemma mkArrow_curry_ty_T : ∀ Gamma ms ts a ,
118     Forall2_T (fun m t ⇒ ty_T Gamma m t) ms ts →
119       ∀ x, ty_T Gamma x (make_arrow_type ts a) →
120         ty_T Gamma (curry x ms) a.
```

Der Beweis für das Lemma in Listing 4.39 folgt direkt aus einer Induktion über der `Forall2_T`-Hypothese.

Listing 4.40: `LongTyping.v` **Siehe Lemma 2.41**

```
129 Lemma long_impl_ty_T : ∀ Gamma m t,
130     long_ty_T Gamma m t → ty_T Gamma m t.
```

Bemerkung: `long_impl_ty_T` verallgemeinert sogar Lemma 2.41, sodass die Aussage für beliebige Typkontexte gilt.

Für den Beweis des Lemmas in Listing 4.40 können nun die Konstruktoren von `long_ty_T` (Listing 4.37) über `make_curry_ty_T` (Listing 4.39) mit `curry` (Listing 4.36) und `make_arrow_type` (Listing 4.33) dargestellt werden.

Um Lemma 2.42 zu beweisen, werden weiterer Hilfslemmata benötigt. Zunächst wird in Listing 4.41 ein Generierungslemma für `ty_T` mit Termen aufgestellt, die durch `curry` (Listing 4.36) generiert sind. Weiter werden Lemmata implementiert, die zeigen, dass eine Typsubstitution nicht beliebig substituieren kann.

Listing 4.41: `Typing.v`

```
143 Lemma mp_gen_T : ∀ Gamma ms x rho,
144     ty_T Gamma (curry (!x) ms) rho →
145       { ts & (Forall2_T (ty_T Gamma) ms ts) *
146              (nth_error Gamma x = Some (make_arrow_type ts
               ↪  rho))}.
```

Um `mp_gen_T` aus Listing 4.41 zu Beweisen, muss eine Induktion *rückwärts* über `ms` durchgeführt werden. Anstelle eines Schrittes `ms ⇝ m : ms` wird ein Schritt `ms ⇝ ms ++ [m]` verwendet. Durch Zurückführen auf das Generierungslemma für die Applikation (Listing 4.31) kann der Beweis für das Lemma erbracht werden.

Falls ein Typ zu einem Typatom substituiert wurde, muss er vorher ebenfalls ein Typatom gewesen sein. Falls er zu einem Pfeiltypen substituiert wurde, war er entweder vorher auch ein Typatom oder die Quelle des substituierten Typen wurde von der Quelle des Ursprungstypen und das Ziel des substituierten Typen von dem Ziel des Ursprungstypen generiert.

Listing 4.42: `Typing.v`

```
157 Lemma subst_var_is_var_T : ∀ Su a rho,
158     ? a = rho.[Su] → { b & rho = ? b }.

164 Lemma subst_arr_is_arr_or_T : ∀ rho sigma Su tau,
165     rho.[Su] = sigma ~> tau →
166       ({sigma' & { tau' & rho = sigma' ~> tau' ∧ sigma'.[Su] =
        ↪  sigma
167               ∧ tau'.[Su] = tau } }) +
168         ({ a & rho = ? a}).
```

Beide Lemmata in Listing 4.42 können durch Fallunterscheidung von `rho` bewiesen werden. Mithilfe der Substitutionslemmata, sowie dem angepassten Generierungslemma (Listing 4.41), kann nun Lemma 2.42 mit einer Induktion über die Länge des Terms bewiesen werden.

Listing 4.43: `LongTyping.v` **Siehe Lemma 2.42**

```
173 Lemma long_general_T : ∀ m Su rho Gamma,
174     ty_T Gamma m rho → long_ty_T Gamma..[Su] m rho.[Su] →
      ↪  long_ty_T Gamma m rho
```

Bemerkung: `Gamma..[Su]` entspricht der Fortsetzung von Substitutionen auf Typkontexten. Siehe hierzu Definition 2.35.

Langableitung für λ-Terme in β-Normalform

Da Aussagen über den Teilformelkalkül mittels dem Datentyp `nfterm` (Listing 4.10) formalisiert sind und es einen starken Zusammenhang zwischen dem Teilformelkalkül und den langtypbaren Termen gibt (siehe Lemma 3.32), muss für diesen Datentyp auch eine Langableitung formalisiert werden.

Listing 4.44: LongTyping.v

```
22 Inductive nfty_long (Gamma : repo) : nfterm → type → Type :=
23   | NFTy_lam_long s sigma tau :
24       nfty_long (sigma :: Gamma) s tau →
25         nfty_long Gamma (\__ s) (sigma ~> tau)
26   | NFTy_var_long :  ∀ x a ts ms
27       (Gammaok : nth_error Gamma x =
28           Some (make_arrow_type ts (? a)))
29       (Lenproof : length ms = length ts),
30         (∀ n (pms : n < length ms),
31           nfty_long Gamma (nth_ok ms n pms)
32             (nth_ok ts n (rew Lenproof in pms))) →
33               nfty_long Gamma (!!x @@ ms) (? a).
```

Die in Listing 4.44 angegebene Formalisierung ähnelt stark `nfty` (Listing 4.34), nur dass die Eigenschaft der Langtypbarkeit erfordert wird, also dass der Zieltyp eines Typen immer ein Typatom ist. Des Weiteren wird im `NFTy_var_long`-Fall kein Beweis gefordert, dass `n < length ts` gilt. Dieser wird mit `rew Lenproof in pms` aus dem Beweis, dass `n < length ms` generiert. Dies erschwert zwar die Lesbarkeit, vereinfacht aber die Handhabung der Konstruktion in weiteren Beweisen.

Da wir über Lemma 3.8 wissen, dass es genügt, prinzipale Inhabitanten in η-Normalform zu finden, kann hier der folgende, alternative Prinzipalitätsbegriff über `nfty_long` angegeben werden.

Listing 4.45: LongTyping.v

```
49 Definition nflong_princ (rho: type) (M: nfterm) : Type :=
50     nfty_long [] M rho * (∀ rho',
51       nfty_long [] M rho' →
52         { Su & rho.[Su] = rho' }).
```

4.4 Teilformelfiltration

Wir betrachten nun die Formalisierung der Teilformelfiltration. Zunächst bedarf es der Formalisierung des Begriffes der relevanten Typen. Ein gegebener Beweisbaum `proof`

kann rekursiv auseinander genommen werden und eine Liste aller verwendeter Typen angelegt werden. Da Beweisbäume immer endlich sind und in jedem Schritt der Baum kleiner wird, terminiert Algorithmus in Listing 4.46.

Listing 4.46: `Filtr.v` **Siehe Definition 3.10**

```
100 Fixpoint TD_f {Gamma m rho} (proof : ty_T Gamma m rho) : list
    ↪  type :=
101   match proof with
102   | Ty_Var _ x rho' eqproof ⇒ [rho']
103   | Ty_Lam _ s sigma tau innerproof ⇒
104       (sigma ~> tau) :: TD_f innerproof
105   | Ty_App _ s t A B proof1 proof2 ⇒
106       B::(TD_f proof1 ++ TD_f proof2)
107   end.
```

Für Definition 3.12 muss über das Enthaltensein von X entschieden werden. Die einfachste Art, in Coq Entscheidbarkeit zu fordern, ist eine Funktion mit Zieltyp `bool` zu fordern. Hierfür kann Menge X über ihre charakteristische Funktion aufgefasst werden, die genau dann `true` ist, wenn ihr übergebener Parameter in X ist. Listing 4.47 zeigt die Implementierung der Filtrationsfunktion.

Listing 4.47: `Filtr.v` **Siehe Definition 3.12**

```
74 Fixpoint filtration
75     (X : type → bool) (a : var) (rho : type) : type :=
76   match rho with
77   | ? b ⇒ ? a
78   | sigma ~> tau ⇒ if (andb (X (sigma ~> tau)) (X tau)) then
79     (filtration X a sigma) ~> (filtration X a tau)
80       else
81     ? a
82   end.
```

Nun bedarf es noch einer Funktion, die genau dann wahr ist, falls ein Typ im Ergebnis von `TD_f` (Listing 4.46) ist. Zunächst muss dafür gezeigt werden, dass das Enthaltensein in `TD_f` entscheidbar ist.

Listing 4.48: `Filtr.v`

```
138 Lemma In_TD_dec {Gamma m rho} : ∀ (deriv : ty_T Gamma m rho)
    ↪  rho',
139     {In rho' (TD_f deriv)} + { ¬ (In rho' (TD_f deriv))}.
```

Auf der Basis des Lemmas aus Listing 4.48 kann nun in Listing 4.49 die folgende Funktion definiert werden.

Listing 4.49: `Filtr.v`

```
145 Definition TD_b {Gamma m rho} (deriv : ty_T Gamma m rho) rho' :
    ↪  bool :=
146     if (In_TD_dec deriv rho') then true else false.
```

Wir können nun in Listing 4.50 eine Formalisierung für Lemma 3.15 angeben.

Listing 4.50: `Filtr.v` **Siehe Lemma 3.15**

```
119 Lemma filter_deriv {m Gamma rho}: ∀ (X : type → bool)
120     (proof : ty_T Gamma m rho),
121   (∀ rho', In rho' (TD_f proof) → X rho' = true) →
122     (∀ a, ty_T (repo_filt X a Gamma) m (filtration X a rho)).
```

Bemerkung:

- `repo_filt` ist die Fortsetzung von `filtration` (Listing 4.47) auf Typkontexten.
- Da der Beweis für das Lemma explizit `(∀ a, ty_T (repo_filt X a Gamma) m (filtration X a rho))` konstruiert, entspricht dies direkt der filtrierten Beweisableitung $\mathcal{F}_X(\mathcal{D})$.

Bevor das notwendige Kriterium aus Lemma 3.18 formuliert und bewiesen werden kann, benötigen wir noch eine Eigenschaft von `filtration`, die einen Zusammenhang zwischen `subformula` (Listing 4.25), `filtration` und Substitution herstellt. Wenn

eine Filtration eines Typen ρ alle Eigenschaften einer Substitution hat, dann haben auch alle Teiltypen der Form $\sigma \to \tau$ von ρ diese Eigenschaft.

Listing 4.51: `Filtr.v`

```
170 Lemma subst_subformula : ∀ sigma tau rho,
171     subformula (sigma ~> tau) rho →
172       ∀ X a Su, rho.[Su] = filtration X a rho →
173         (sigma ~> tau).[Su] = filtration X a (sigma ~> tau).
```

Im Beweis des Lemmas in Listing 4.51 wird eine Induktion über `subformula` durchgeführt. Der reflexive Fall ist trivial, die beiden Pfeilfälle lassen sich über eine Fallunterscheidung über dem Enthaltensein in X beweisen.

Nun kann die Implementierung von Lemma 3.18 angeben werden.

Listing 4.52: `Filtr.v` **Siehe Lemma 3.18**

```
201 Lemma filter_princ_nec : ∀ m rho (D : ty_T [] m rho) sigma tau,
202     subformula (sigma ~> tau) rho → (TD_b D tau = false) →
      ↪  princ rho m →
203       False.
```

Um das Lemma in Listing 4.52 zu beweisen, wird zunächst `filter_deriv` (Listing 4.50) genutzt, um aus `D` eine gefilterte Ableitung `filtD` zu generieren. Über `princ` (Listing 4.30) wissen wir, dass es für `filtD` eine Substitution für ρ geben muss, die der `filtration` von ρ zu `TD_b D` entspricht. Durch `subst_subformula` (Listing 4.51) wissen wir, dass das auch für `sigma ~> tau` gelten muss. Dies steht jedoch im Widerspruch zur Filtration und `TD_b` (Listing 4.49), da τ nicht in X ist.

4.5 Teilformelkalkül

Der Teilformelkalkül führt über Definition 3.21 eine weitere Ableitung ein. Für die Formalisierung werden wir ausschließlich die Terme in Normalform (Siehe Listing 4.10) betrachten. Dies vereinfacht Fallunterscheidungen und erlaubt es uns direkt die i-te Komponente einer Applikation auf einen Pfad $\pi \cdot \texttt{Tgt}^{i-1}$ abzubilden. Wir werden zunächst die grundlegenden Konstruktionen und allgemeinene Hilfslemmata betrachten, die für den Beweis für Lemma 3.32 notwendig sind. Im Zuge dessen werden wir die

Entscheidbarkeit des transitiv-symmetrischen Abschlusses über die Pfadrelationen zeigen. Dies ist besonders für Lemma 3.33 wichtig. Danach werden wir explizit die Beweise für Lemma 3.24 und Korollar 3.25 betrachten. Anschließend werden wir Algorithmus 2 implementieren sowie grundlegende Aussagen darüber beweisen, um daraufhin die Formalisierung von Lemma 3.32 zu beweisen. Abschließend werden wir Lemmata 3.33 und 3.34, sowie Lemma 3.38 formalisieren, sowie die Ersetzung aus Lemma 3.33 implementieren. Die letzten drei Lemmata werden wir jedoch hier nicht mehr beweisen.

4.5.1 Definition der Pfadrelation

Der Teilformelkalkül ähnelt in vielerlei Hinsicht der Langableitung. Ein großer Unterschied ist, dass die $(\text{varapp})_R$-Regel eine Nebenbedingung in der Form einer Relation R hat. Wir benötigen somit eine Repräsentation der Relation R. Eine einfache Möglichkeit wäre es, diese als `list (path * path)` zu implementieren. Die Bedingung $(\pi, \pi') \in R$ übersetzt sich dann direkt in die Aussage `In (pi, pi') R`.

Sowohl die Konstruktion einer Relation in Lemma 3.28, sowie in Definition 3.30 nutzen den transitiv-symmetrischen Abschluss einer kleineren, einfach zu generierenden Relation. Den transitiven-symmetrischen Abschluss einer Relation zu berechnen, die über eine Liste implementiert ist, ist jedoch verhältnismäßig schwierig.

Rekursiv definierte Funktionen in Coqwerden üblicherweise über das Schlüsselwort `Fixpoint` definiert. Diese Funktionen müssen immer beweisbar terminieren. Dies wird dadurch erzwungen, dass eine Funktion einen Parameter als Hauptstruktur ansieht und auf dieser Struktur in jedem Rekursionsschritt strukturell kleiner werden muss.

Eine Funktion, die den transitiv-symmetrischen Abschluss einer Relation berechnet, hat den mit Typen `list (A * A)` $\rightarrow$ `list (A * A)`. Der einzige Parameter, der hier angegeben ist, ist die Relation selbst. Da der transitiv-symmetrische Abschluss immer mindestens die Ursprungsrelation beinhaltet, wird hier die Relation R nicht strukturell kleiner. Es bedarf somit eines anderen Ansatzes. Anstelle das Ergebnis des Abschlusses als neue Liste konstruieren, geben wir in Listing 4.53 ein Konstrukt an, das den Abschluss nur beschreibt.

Listing 4.53: Utils.v

```
778 Inductive ts_cl_list {A} (R: list (A * A)) : A → A → Type :=
779   | ts_R_list : ∀ a b, In (a, b) R → ts_cl_list R a b
780   | ts_symm_list : ∀ a b, ts_cl_list R a b → ts_cl_list R b a
781   | ts_trans_list : ∀ a b c,
782       ts_cl_list R a b → ts_cl_list R b c → ts_cl_list R a c.
```

Die verwendeten Relationen R_M und R_τ können nun zunächst als Liste generieren werden und über diese Konstruktion transitiv und symmetrisch abgeschlossen werden. Der allgemeine Typ für die Relationen ist somit `A → A → Type`. Betrachten wir die Lemmata 3.32 und 3.34, stellen wir fest, dass eine wichtige Relation zwischen diesen Relationen die Teilmengenrelation ist. Diese ist in Listing 4.54 einmal für die Darstellung in `Type` sowie für die Darstellung als Liste definiert.

Listing 4.54: Utils.v

```
1606 Definition Rsub {A} (R R' : A → A → Type) := ∀ (pi pi' : A),
1607     R pi pi' → R' pi pi'.

1608 Definition Rsub_list {A} R R' := ∀ (pi pi' : A),
1609     In (pi, pi') R → In (pi, pi') R'.
```

4.5.2 Entscheidbarkeit des transitiv-symmetrischen Abschlusses

Eine wichtige Eigenschaft, die diese Relationen haben müssen, ist, entscheidbar zu sein. Betrachten wir die Ersetzung in Lemma 3.33. Diese ersetzt einen Pfad in einem Typen abhängig davon, ob der Pfad in der Relation R_M enthalten ist. Damit die Ersetzung konstruiert werden kann, muss das Enthaltensein von Pfaden in R entscheidbar sein. Wenn wir direkt die eine auf Listen basierende Konstruktion verwenden, ist dies trivial. Falls ein Typ `A` die Klasse `EqDec` implementiert – also es entscheidbar ist, ob für `a b : A` gilt `a == b` –, so ist `In a X` für alle `a : A` und `X : list A` entscheidbar. Dies ist bereits in der Standardbibliothek von Coq mittels `in_dec` bewiesen. Es bleibt somit zu zeigen, dass auch der transitiv-symmetrische Abschluss entscheidbar ist. Hierfür zerlegen wir den transitiv-symmetrischen Abschluss in seinen transitiven und seinen symmetrischen Teil und zeigen die Entscheidbarkeit der einzelnen Abschlüsse.

Der symmetrische Teil lässt sich direkt als Liste implementieren, indem eine weitere Liste generiert wird, in der alle Tupel umgedreht werden, und diese mit der Ursprungsliste kombiniert wird.

Listing 4.55: `Utils.v`

```
819 Definition sym_hull_list {A} (R: list (A * A)) : list (A * A) :=
820     R ++ flipped R.
```

Bemerkung: `flipped` dreht alle Tupel von `R` um.

Über `in_dec` wissen wir, dass das Enthaltensein im symmetrische Abschluss entscheidbar ist, wenn `A` die Klasse `EqDec` implementiert.

Betrachten wir nun den transitiven Abschluss. Hier wird keine Funktion angegeben, die explizit den Abschluss berechnet; stattdessen wird dieser mit der Konstruktion in Listing 4.56 beschrieben.

Listing 4.56: `Utils.v`

```
927 Inductive trans_hull {A} (R : list (A * A)) : A → A → Type :=
928   | trans_R : ∀ a b, In (a, b) R → trans_hull R a b
929   | trans_trans : ∀ s x t, trans_hull R s x → trans_hull R x t
      ↪  →
930     trans_hull R s t.
```

Für die Betrachtung des transitiven Abschlusses bietet es sich an, die Relation R als Kantenmenge eines Graphen aufzufassen. Das Enthaltensein eines Tupels (s, t) im Abschluss gleicht dann der Erreichbarkeit von t von s aus. Der erste Ansatz hierfür war es, auszunutzen, dass, falls es einen Pfad zwischen s und t gibt, es auch einen Pfad der Länge $|R|$ gibt, und dann per Induktion über die Länge des Pfades die Entscheidbarkeit zeigen. Es stellt sich jedoch heraus, dass es ausreicht, sich die besuchten Kanten zu merken. Wir betrachten hierfür die Konstruktion in Listing 4.57.

Listing 4.57: `Utils.v`

```
1147 Inductive t_path {A} (R: list (A *A)) :
1148     list (A * A) → A → A → Type :=
1149   | t_path_R : ∀ a b, In (a, b) R → t_path R [(a, b)] a b
1150   | t_path_trans : ∀ s x t p, In (s, x) R → t_path R p x t →
1151       t_path R ((s,x) :: p) s t.
```

Der größte Unterschied zu `trans_hull` (Listing 4.56) ist, dass die Transitivität schrittweise erreicht wird. Die Erreichbarkeit Zweier Knoten s und t lässt sich nun nicht mehr in die Erreichbarkeit von s zu einem beliebigen x und die Erreichbarkeit von x zu t zeigen. Falls t von x aus erreichbar ist, muss die Kante (s, x) in dieser Implementierung direkt in der Relation vorhanden sein, damit t von s aus erreichbar ist. Die Kante (s, x) wird dann direkt dem Pfad hinzugefügt.

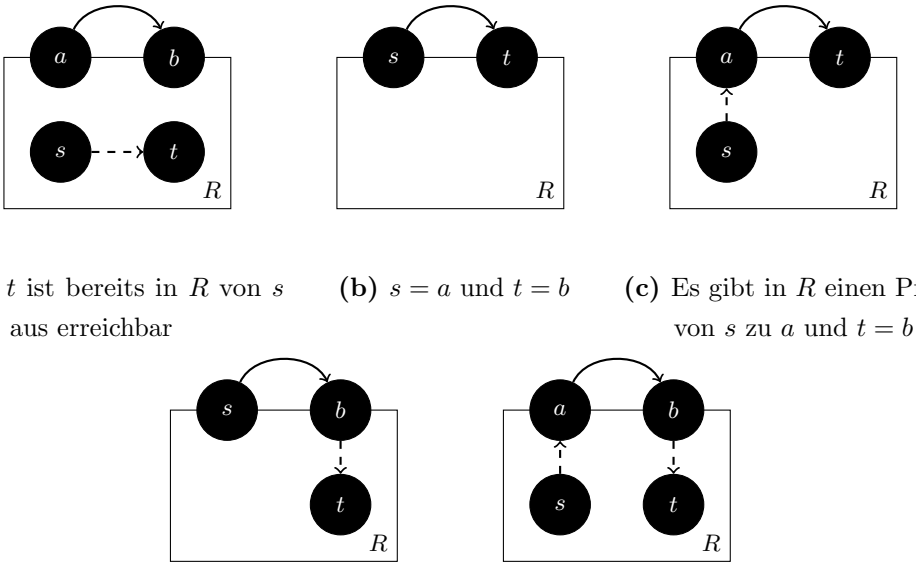
Es lässt sich zeigen, dass dies nicht die Erreichbarkeit von t von s aus beeinträchtigt, und so `trans_hull` entscheidbar ist, wenn `t_path` entscheidbar ist. Wir betrachten hierzu Überführungen der beiden Konstruktionen ineinander.

Listing 4.58: `Utils.v`

```
1205 Lemma t_path_ex {A} R (s t : A) : trans_hull R s t →
1206     {P & t_path R P s t}.

1215 Lemma t_path_trh {A} R (s t: A) P :
1216     t_path R P s t → trans_hull R s t.
```

Für die Hinrichtung der in Listing 4.58 definierten Lemmata lässt sich der Pfad `P` über die in `trans_hull` (Listing 4.56) verwendeten `trans_trans`-Regeln aufbauen. Für die Rückrichtung kann der Pfad einfach ignoriert werden. Es bleibt somit zu zeigen, dass `t_path` entscheidbar ist. Hierfür bedarf es noch einiger Hilfslemmata. Zunächst können wir feststellen, dass `t_path` monoton im Bezug auf R ist. Mehr Kanten verhindern nicht, dass ein Pfad existiert. Dies kann mit dem folgenden Lemma in Listing 4.59 formalisiert werden.



(a) t ist bereits in R von s aus erreichbar

(b) $s = a$ und $t = b$

(c) Es gibt in R einen Pfad von s zu a und $t = b$

(d) Es gibt in R einen Pfad von b zu t und $s = a$

(e) Es gibt in R einen Pfad von s zu a und einen Pfad von b zu t.

Abbildung 4.2: Erreichbarkeit im Graphen $(a, b) :: R$

Listing 4.59: `Utils.v`

```
1222 Lemma t_path_pump {A} R (s t : A) P : t_path R P s t →
1223     ∀ ab, t_path (ab::R) P s t.
```

Weiter können wir durch Analyse der Erreichbarkeit von zwei Knoten s und t in einer Relation $(a, b) :: R$, eine notwendige Bedingung formulieren, nach der t von s aus erreichbar ist. Die Erreichbarkeit lässt sich erschöpfend durch die folgenden fünf Fälle abdecken. Diese sind ebenfalls in Abbildung 4.2 dargestellt.

(a) t ist bereits in R von s aus erreichbar

(b) $s = a$ und $t = b$

(c) Es gibt in R einen Pfad von s zu a und $t = b$

(d) Es gibt in R einen Pfad von b zu t und $s = a$

(e) Es gibt in R einen Pfad von s zu a und einen Pfad von b zu t.

Dies können wir im Form des folgenden Lemmas in Listing 4.60 formalisieren.

Listing 4.60: `Utils.v`

```
1265 Lemma t_path_trans_R_and {A} {eqdec: EqDec A eq} :
1266     ∀ R (s t a b : A) P',
1267       t_path ((a, b)::R) P' s t →
1268         {P & t_path R P s t} +
1269         {(a, b) = (s, t)} +
1270         {P & prod (t_path R P s a) (t = b)} +
1271         {P & prod (t_path R P b t) (s = a)} +
1272         {P1 & {P2 & prod (t_path R P1 s a) (t_path R P2 b t)}}.
```

Mithilfe von `t_path_trans_R_and` können wir nun die Entscheidbarkeit von `t_path` beweisen.

Listing 4.61: `Utils.v`

```
1301 Lemma t_path_dec {A} {eqdec : EqDec A eq} : ∀ R (s t : A),
1302     {P & t_path R P s t} + {∀ P, (t_path R P s t → False)}.
```

Das Lemma in Listing 4.61 kann nun durch Induktion über R bewiesen werden. Der Fall für die leere Relation gilt trivialerweise, t ist nicht von s aus erreichbar. Im Induktionsschritt $R \rightsquigarrow (a, b) :: R$ kann nun jeder Fall aus Abbildung 4.2 getestet werden und, falls keiner dieser Fälle zutrifft, kann mittels `t_path_trans_R_and` (Listing 4.60) gezeigt werden, dass `t` von `s` aus nicht erreichbar ist.

Durch die Entscheidbarkeit der Pfadkonstruktion und den Überführungen `t_path_ex` (Listing 4.58) und `t_path_trh` (Listing 4.58) lässt sich direkt die Entscheidbarkeit des transitiven Abschlusses zeigen.

Listing 4.62: `Utils.v`

```
1331 Lemma trans_hull_dec {A} {eqdec: EqDec A eq}: ∀ R (s t :A),
1332     trans_hull R s t + (trans_hull R s t → False).
```

Es bleibt nun noch zu zeigen, dass aus der Entscheidbarkeit des transitiven Abschlusses und der Entscheidbarkeit der symmetrischen Abschlusses die Entscheidbarkeit des

transitiv-symmetrischen Abschlusses folgt. Dafür zeigen wir zunächst die Äquivalenz zwischen `trans_hull` (Listing 4.56) angewendet auf `sym_hull_list` (Listing 4.55) und `ts_cl_list` (Listing 4.53).

Listing 4.63: `Utils.v`

```
1056 Lemma ts_cl_list_trans_sym {A} {eqdec: EqDec A eq} :
1057     ∀ R (s t : A), ts_cl_list R s t →
1058       trans_hull (sym_hull_list R) s t.

1075 Lemma trans_sym_ts_cl_list {A} {eqdec: EqDec A eq} :
1076     ∀ R (s t : A), trans_hull (sym_hull_list R) s t →
1077       ts_cl_list R s t.
```

Über die Überführung aus Listing 4.63 sowie die Entscheidbarkeit von `t_path` lässt sich direkt die Entscheidbarkeit des transitiv-symmetrischen Abschlusses zeigen.

Listing 4.64: `Utils.v`

```
1339 Lemma ts_cl_list_dec {A} {eqdec: EqDec A eq}: ∀ R (s t: A),
1340     ts_cl_list R s t + (ts_cl_list R s t → False).
```

Zuletzt zeigen wir noch, dass der transitiv-symmetrischen Abschluss die Eigenschaft der *inneren* Reflexivität erfüllt. Siehe hierzu auch die Bemerkung zu Definition 3.30.

Listing 4.65: `Utils.v`

```
2075 Lemma almost_refl_l {A} R : ∀ (a b :A), ts_cl_list R a b →
2076     ts_cl_list R a a.

2081 Lemma almost_refl_r {A} R : ∀ (a b :A), ts_cl_list R a b →
2082     ts_cl_list R b b.
```

Der Beweis der beiden Aussagen aus Listing 4.65 folgt direkt durch die folgenden Implikationen:

$$(a,b) \in R \overset{\text{Sym.}}{\Rightarrow} (a,b),(b,a) \in R^{ts} \overset{\text{Trans.}}{\Rightarrow} (a,a),(b,b) \in R^{ts}$$

4.5.3 Ableitung und Teilableitung

Um nun den Teilformelkalkül definieren zu können, implementieren wir zunächst in Listing 4.66 eine Funktion, die uns Pfade in der Form $\pi \cdot \texttt{Tgt}^{i-1}$ generiert, wie sie im applikativen Fall in Definition 3.21 verwendet werden.

Listing 4.66: `Paths.v`

```
261 Definition make_tgt_path (pi: path) (n : nat) :=
262     pi ++ (repeat Tgt n) ++ [Src].
```

Nun kann die Ableitung des Teilformelkalküls definiert werden. Auch diese ähnelt stark `nfty_long` (Listing 4.44) und `nfty` (Listing 4.34).

Listing 4.67: `SfC.v` **Siehe Definition 3.21**

```
20 Inductive SfC (Delta : list path) (R: path → path → Type) :
21     nfterm → path → Type :=
22   | SfC_I s pi : SfC ((pi ++ [Src]) :: Delta) R s (pi ++ [Tgt])
     ↪  →
23       SfC Delta R (\__ s) pi
24   | SfC_E ms pi pi' x : nth_error Delta x = Some pi →
25       R (pi ++ repeat Tgt (length ms)) pi'  →
26         (∀ n (p: n < length ms),
27             SfC Delta R (nth_ok ms n p) (make_tgt_path pi n) )
               ↪  →
28           SfC Delta R (!! x @@ ms) pi'.
```

Da wir für einige Beweise explizit Teilableitungen von `SfC` betrachten müssen, führen wir in Listing 4.68 eine Relation ein, die Teilableitungen beschreibt.

Listing 4.68: `SfC.v`

```
304 Inductive SfC_subj R : ∀ Delta Delta' m m' pi pi',
305     SfC Delta R m pi → SfC Delta' R m' pi' → Type :=
306   | SfC_subj_refl : ∀ Delta m pi (proof: SfC Delta R m pi),
307       SfC_subj _ _ _ _ _ _ _ proof proof
```

```
308  | SfC_subj_trans : ∀ Delta Delta' Delta'' m m' m'' pi pi' pi''
309      (proof1 : SfC Delta R m pi)
310      (proof2: SfC Delta' R m' pi')
311      (proof3: SfC Delta'' R m'' pi''),
312        SfC_subj _ _ _ _ _ _ _ proof1 proof2 →
313          SfC_subj _ _ _ _ _ _ _ proof2 proof3 →
314            SfC_subj _ _ _ _ _ _ _ proof1 proof3
315  | SfC_subj_I : ∀ Delta m pi (proof: SfC ((pi ++ [Src]) ::
     ↪  Delta)
316      R m (pi ++ [Tgt])),
317        SfC_subj _ _ _ _ _ _ _ proof (SfC_I _ _ _ _ proof)
318  | SfC_subj_E : ∀ Delta pi pi' ms x
319      (deltaok: nth_error Delta x = Some pi)
320      (Rproof: R (pi ++ repeat Tgt (length ms)) pi')
321      (proofs: (∀ n (p: n < length ms),
322        SfC Delta R (nth_ok ms n p) (make_tgt_path pi n) ))
323      n (ltproof: (n < length ms)),
324    SfC_subj _ _ _ _ _ _ _ (proofs n ltproof)
325       (SfC_E _ _ _ _ _ _ deltaok Rproof proofs).
```

Während `SfC_subj_refl` und `SfC_subj_trans` den reflexiven bzw. transitiven Abschluss einer Teilableitung beschreiben, beschreiben `SfC_subj_I` und `SfC_subj_E` genau einen Schritt einer Ableitung. Die übergebenen Argumente der Konstruktoren der Teilableitung werden lediglich an die Konstruktoren des Teilformelkalküls (Listing 4.67) weitergeleitet.

Des Weiteren benötigen wir für den Beweis von Lemma 3.24, eine Möglichkeit Aussagen über jeden einzelnen Schritt einer Ableitung zu formulieren. Hierfür formulieren wir in Listing 4.69 eine Funktion, die eine Aussage `P` über einzelne Ableitungsschritte und einer Pfadableitung `proof` entgegennimmt und eine Konjunktion über alle Schritte der Ableitung generiert.

Listing 4.69: SfC.v

```
357 Fixpoint each_judg_SfC {Delta R m pi}
358   (P : ∀ Delta' R' m' pi', SfC Delta' R' m' pi' → Prop)
359   (proof : SfC Delta R m pi) : Prop :=
360     P Delta R m pi proof ∧
361       match proof with
362       | SfC_I _ _ s' pi' proof' ⇒ each_judg_SfC (P) proof'
363       | SfC_E _ _ ms pi pi' x DeltaProof Rproof proof' ⇒
364           ∀ (n : nat) (p : n < length ms),
365             each_judg_SfC (P) (proof' n p)
366       end.
```

Da `each_judg_SfC` Aussagen über alle Teilableitungen einer Ableitung trifft, lässt sich ein Zusammenhang zu Konstruktion `SfC_subj` (Listing 4.68) herstellen.

Listing 4.70: SfC.v

```
366 Lemma each_judg_subj_SfC : ∀ Delta R t pi (m : SfC Delta R t pi)
    ↪  P,
367     each_judg_SfC P m →
368       ∀ Delta' t' pi' (n : SfC Delta' R t' pi'),
369         SfC_subj _ _ _ _ _ _ _ n m →
370           (each_judg_SfC P n).


381 Lemma each_judg_subj_SfC_P : ∀ Delta R t pi (m : SfC Delta R t
    ↪  pi) P,
382     each_judg_SfC P m →
383       ∀ Delta' t' pi' (n : SfC Delta' R t' pi'),
384         SfC_subj _ _ _ _ _ _ _ n m →
385           P _ _ _ _ n.
```

Bemerkung: Die beiden Lemmata in Listing 4.70 zeigen sehr ähnliche Aussagen, die an unterschiedlichen Stellen benötigt werden. `each_judg_SfC` zeigt die allgemeinere Aussage, dass, wenn für zwei Ableitungen `m` und `n` gilt, dass `n` eine Teilableitung nach `SfC_subj` (Listing 4.68) von `m` ist und eine Aussage `P` für `m` und alle Teilableitngen nach `each_judg_SfC` (Listing 4.69) gilt, dann gilt sie auch für alle Teilableitungen von `n`.
Das zweite Lemma ist ein Spezialfall des ersten und sagt lediglich aus, dass in diesem Fall die Aussage für `n` gilt.

Das erste Lemma in Listing 4.70 kann durch Induktion über `SfC_subj` (Listing 4.68) bewiesen werden. Das zweite Lemma wird mit dem ersten bewiesen.

Mittels des in Listing 4.71 aufgestellten Lemmas kann eine Implikation von Aussagen P und Q auf Einzelschritten einer Ableitung auf die Implikation der Aussagen auf dem gesamten Ableitungsbaum geliftet werden. Dies wird für den Zusammenhang zwischen Lemma 3.24 und Korollar 3.25 benötigt.

Listing 4.71: `SfC.v`

```
459 Lemma each_judg_impl : ∀
460     (P : ∀ Delta R m pi, SfC Delta R m pi → Prop)
461     (Q : ∀ Delta R m pi, SfC Delta R m pi → Prop),
462   (∀ Delta R m pi (proof : SfC Delta R m pi),
463       P _ _ _ _ proof →  Q _ _ _ _ proof) →
464     ∀ Delta R m pi (proof : SfC Delta R m pi),
465       each_judg_SfC P proof → each_judg_SfC Q proof.
```

Für den Beweis von Lemma 3.32 wird ein Generierungslemma des Teilformelkalküls benötigt. Intuitiv ist klar, dass ein `nfterm` (Listing 4.10) der Form `\__ m` mit dem Konstruktor `SfC_I` (Listing 4.67) und ein `nfterm` der Form `!! x @@ ms` mit dem Konstruktor `SfC_E` bewiesen wird. Mit den Lemmata aus Listing 4.72 kann diese Intuition formalisiert werden.

Listing 4.72: `SfC.v`

```
90 Lemma SfC_gen_app : ∀ R x ms pi' Delta
91     (proof : SfC Delta R (!! x @@ ms) pi'),
92   SfC_E Delta R ms
93     (get_subproof_app_pi proof) pi' x
94       (get_subproof_app_deltaok proof)
95         (get_subproof_app_Rok proof)
96           (get_subproof_app proof) = proof.

214 Lemma SfC_gen_lam R m pi Delta (proof : SfC Delta R (\__ m) pi)
    ↪  :
215     SfC_I Delta R m pi (get_subproof_lam proof) = proof .
```

Bemerkung:

- Die mit `get_subproof_` gepräfixten Funktionen extrahieren aus einem gegebenen Beweis genau die entsprechenden Teilbeweise.
- Für den Beweis dieser beiden Lemmata war es notwendig, einen großen Teil des Beweises explizit als λ-Term anzugeben (Siehe Abschnitt 2.3.1), da die Taktiksprache von Coq keinen entsprechenden Beweis formulieren konnte. Dadurch ist der implementierte Beweis sehr umständlich und schwer nachzuvollziehen. Im Kern wird lediglich eine Fallunterscheidung über `proof` durchgeführt und ein Fall wird ad absurdum geführt.

4.5.4 Eigenschaften der Pfadrelationen

Wir betrachten nun die Eigenschaften, die Pfadrelationen haben. Insbesondere werden wir dabei Korollar 3.25 formalisieren und beweisen. Zunächst formulieren wir in Listing 4.73 auf Basis von `Rsub` (Listing 4.54) das Monotonielemma 3.22, das später benötigt wird, um die Korrektheit der Implementierung von Algorithmus 2 zu beweisen. Analog zu `Rsub` formulieren wir auch ein Lemma für die Behandlung als Liste.

Listing 4.73: `SfC.v` **Siehe Lemma 3.22**

```
321 Lemma sfc_monotone_aux : ∀ m R R' pi Delta, Rsub R R' →
322     SfC Delta R m pi → SfC Delta R' m pi.

334 Lemma sfc_monotone : ∀ m R R', Rsub R R' →
335     SfC [] R m [] → SfC [] R' m [].

342 Lemma sfc_monotone_aux_list : ∀ m R R' pi Delta, Rsub_list R R'
    ↪  →
343     SfC Delta (ts_cl_list R) m pi →
344       SfC Delta (ts_cl_list R') m pi.
```

Bemerkung:

- `sfc_monotone_aux` ist die verallgemeinerte Variante von `sfc_monotone` für beliebige Pfadkontexte Δ.
- Für die Listendarstellung ist nur das verallgemeinerte Lemma implementiert. Eine Variante für den leeren Pfadkontext wird nicht benötigt.

`sfc_monotone_aux` (Listing 4.73) lässt sich direkt über Induktion von `SfC Delta R m pi` beweisen. `sfc_monotone` sowie `sfc_monotone_aux_list` können direkt über das erste Lemma bewiesen werden.

Im Folgenden werden wir Korollar 3.25 formulieren und für den Beweis die notwendigen Lemmata aufstellen. Zunächst stellen wir die Bedingung, die in Lemma 3.24 verwendet wird, für einen einzelnen gegebenen Ableitungschritt auf. Die hierfür notwendigen Teilbedingungen, dass einerseits die Anzahl an `Src` gerade in π und dass die Anzahl an `Src` ungerade in einem Pfad im Kontext ist, werden in Listing 4.74 formalisiert.

Listing 4.74: `Paths.v`

```
264 Definition even_ones pi := Nat.Even (count_occ dir_eqdec pi
    ↪  Src).
```

```
274 Definition odd_repo (Delta : list path) :=
275     Forall (fun pi ⇒ Nat.Odd (count_occ dir_eqdec pi Src))
     ↪  Delta.
```

Nun kann eine Bedingung für Ableitungsschritte von `SfC` (Listing 4.67) angegeben werden, welche die einzelnen Bedingungen aus Listing 4.74 vereint.

Listing 4.75: `SfC.v`

```
383 Definition evenodd_cond {Delta R m pi} (proof : SfC Delta R m
    ↪  pi) :=
384     (even_ones pi) ∧ (odd_repo (Delta)).
```

Um nun zu fordern, dass jeder Ableitungsschritt einer Ableitung diese Bedingung erfüllt, kann die Konstruktion `each_judg_SfC` (Listing 4.69) genutzt werden, und so Lemma 3.24 in Listing 4.76 formalisiert werden.

Listing 4.76: `SfC.v` **Siehe Lemma 3.24**

```
423 Lemma evenodd {R m} (proof : SfC [] R m []) :
424     each_judg_SfC (@evenodd_cond) proof.
```

Der Beweis hierfür folgt analog zu dem Beweis in Lemma 3.24 über die in Listing 4.77 formalisierte allgemeinere Aussage.

Listing 4.77: `SfC.v`

```
400 Lemma evenodd_aux {Delta R m pi} (proof : SfC Delta R m pi) :
401     evenodd_cond proof → each_judg_SfC (@evenodd_cond) proof.
```

Dies kann durch Induktion über `proof` und Auflösen der `each_judg_SfC` Konstruktion sowie einiger Regeln über gerade und ungerade Pfade bewiesen werden.

Die Bedingung aus Korollar 3.25, dass kein Ableitungsschritt die Bedingung $(\pi, \pi) \in R$ hat, wird ebenfalls zunächst auf einem einzelnen Ableitungsschritt formuliert, um dann mit `each_judg_SfC` auf die ganze Ableitung geliftet zu werden. Wir betrachten hier die Konstruktoren von `SfC` (Listing 4.67) einzeln. Der Konstruktor für die

Applikation stellt keine Anforderungen an `R`, somit kann in diesem Fall immer `True` zurückgegeben werden. Für den applikativen Fall reicht es zu fordern, dass $\pi' \neq \pi \cdot \texttt{Tgt}^n$ gilt.

Listing 4.78: `SfC.v`

```
Definition r_not_refl_cond {Delta R m pi} (proof : SfC Delta R m
↪  pi)  :=
 match proof with
 | SfC_I _ _ _ _ _ ⇒ True
 | SfC_E _ _ ms pi pi' _ _ _ _ ⇒
     ¬ (pi ++ repeat Tgt (length ms) = pi')
 end.
```

Wir können nun zeigen, dass aus `evenodd_cond` (Listing 4.75) die Bedingung `r_not_refl_cond` (Listing 4.78) folgt.

Listing 4.79: `SfC.v`

```
Lemma evenodd_2_r_not_refl {Delta R m pi} (proof : SfC Delta R m
↪  pi) :
   evenodd_cond proof → r_not_refl_cond proof.
```

Mittels `each_judg_impl` (Listing 4.71) kann `evenodd_2_r_not_refl` (Listing 4.79) auf die gesamte Ableitung geliftet werden. Da `evenodd` (Listing 4.76) gilt, gilt auch das folgende Lemma.

Listing 4.80: `SfC.v` **Siehe Korollar 3.25**

```
Lemma r_not_refl {R m} (proof : SfC [] R m []) :
   each_judg_SfC (@r_not_refl_cond) proof.
```

4.5.5 Verifikation des Algorithmus R_M-aux

In diesem Abschnitt werden wir die Implementierung des Algorithmus 2 betrachten sowie seine Korrektheit und Minimalität beweisen. Hierfür betrachten wir zunächst die Implementierung des Algorithmus in Listing 4.81.

Listing 4.81: `SfC.v` **Siehe Algorithmus 2**

```
491 Fixpoint R_m_aux (Delta: list path) (pi': path) (m : nfterm)
492   {struct m} : option (list (path * path)) :=
493     match m with
494     | \__ s ⇒ R_m_aux ((pi' ++ [Src]) :: Delta) (pi' ++ [Tgt])
      ↪  s
495     | !! x @@ ms ⇒
496         match nth_error Delta x with
497         | None ⇒ None
498         | Some pi ⇒
499            option_concat (combine_with
500             (fun x n ⇒ R_m_aux Delta (make_tgt_path pi n) x)
501             ms (range (length ms)) ++
502              [Some [((pi ++ repeat Tgt (length ms), pi'))]])
503         end
504      end.
```

Bemerkung:

- `option_concat`, `combine_with` sowie `range` sind in `Utils.v` definierte Hilfsfunktionen.
  - `option_concat : list (option (list A)) → option (list A)` gibt `None` zurück, wenn mindestens ein Element der Eingabe `None` ist. Ansonsten werden alle Listen konkateniert und im `option` Datentyp gekapselt.
  - `combine_with (f: (A → B → C)) : list A → list B → list C` arbeitet wie `combine`, nur, dass nicht eine Liste von Tupeln `(A * B)` generiert wird, sondern jedes Element der Rückgabeliste von der Funktion `f` generiert wird.
  - `range n : list nat` ist direkt über `seq 0 n` definiert.
- In `SfC.v` wird anstelle der Funktion `combine_with` aus `Utils.v` ein Fixpunkt *inline* mit dem Namen `combine_with` definiert, der sich genau so verhält. Dies muss auf diese Weise gelöst werden, damit Coq er-

kennt, dass im Rekursionsschritt `m` strukturell kleiner wird. Das Lemma `combine_with_inline` zeigt die Gleichheit der beiden Formulierungen.

Des Weiteren betrachten wir hier noch einige Definitionen, die den Aufruf an `R_m_aux` kapseln. Zum einen benötigen wir einen initalen Aufruf ohne Kontext, zum anderen benötigen wir den transitiv-symmetrische Abschluss mittels `ts_cl_list` (Listing 4.53). Bei letzterem ist zu beachten, dass `R_m_aux None` zurückgibt, falls der Algorithmus fehlschlägt. In diesem Fall interpretieren wir den Abschluss als Funktion, die immer `False` zurückgibt.

Listing 4.82: `SfC.v`

```
510 Definition R_m m := R_m_aux [] [] m.

514 Definition R_m_ts m := match R_m m with
515   | None ⇒ fun pi pi' ⇒ False
516   | Some Rmm ⇒ ts_cl_list (Rmm)
517 end.
```

Wir beweisen nun die Korrektheit des Algorithmus aus, also dass, falls der Algorithmus erfolgreich terminiert, die über `R_m_aux` (Listing 4.81) generierten Relationen, zu einer korrekten Typableitung führt.

Listing 4.83: `SfC.v` **Siehe Lemma 3.28**

```
567 Lemma R_m_ts_correct : ∀ m Rm Delta pi,
568     R_m_aux Delta pi m = Some Rm  →
569       SfC Delta (ts_cl_list Rm) m pi.
```

Das in Listing 4.83 formulierte Lemma kann durch Induktion über `m` bewiesen werden. Der Abstraktionsfall ist trivial, da hier die Relation nicht genutzt wird. Im Applikationsfall kann dadurch, dass `R_m_aux Delta pi m = Some Rm` gefordert wird, die durch `option_concat` und `combine_with` konstruierte Liste auseinander genommen werden und immer gezeigt werden, dass niemals `None` das Ergebnis eines rekursiven Aufrufs war. Die Applikation kann dann durch den Konstruktor von `SfC` (Listing 4.67) auseinander genommen werden und es kann gezeigt werden, dass das

geforderte Tupel `(pi ++ repeat Tgt (length ms), pi')` im entsprechenden Schritt hinzugefügt wird.

Da der Induktionschritt über `nfterm` (Listing 4.10) durchgeführt wird, wird zwar `m` destruiert, die Pfadrelation wird jedoch nicht für den induktiven Abstieg angepasst. Hier wird `sfc_monotone` (Listing 4.73) genutzt, um zu zeigen, dass eine Ableitung auch mit einer größeren Relation durchführbar ist.

Es bleibt zu zeigen, dass die von `R_m_aux` generierte Relation auch die kleinste Relation ist.

Listing 4.84: `SfC.v` **Siehe Definition 3.27 und Lemma 3.28**

```
805 Lemma R_m_ts_minimal : ∀ m Rm Delta pi,
806     SfC Delta Rm m pi → ∀ Rm', R_m_aux Delta pi m = Some Rm' →
807       Rsub (fun p p' ⇒ In (p, p') Rm') Rm.
```

Dies kann direkt durch Induktion über `SfC` bewiesen werden. Der Abstraktionsfall ist hier ebenfalls trivial. Im Applikationsfall kann analog zu `R_m_ts_correct` die Liste der Subaufrufe auseinandergenommen werden, da gefordert ist, dass `R_m_aux` erfolgreich terminiert.

Sowohl `R_m_ts_correct` als auch `R_m_ts_minimal` fordern, dass `R_m_aux` (Listing 4.81) erfolgreich terminiert. Wir werden hier abschließend noch die hinreichende Bedingung aus Lemma 3.26 formalisieren, nach der `R_m_aux` erfolgreich terminiert, wenn der entsprechende `nfterm closed` (Listing 4.13) ist. Dies kann über die allgemeinere Aussage bewiesen werden, dass `R_m_aux` nicht fehlschlägt, wenn der übergebene Kontext allen freien Variablen einen Pfad zuweist. Hierfür wird `all_var_in_repo` (Listing 4.35) genutzt.

Listing 4.85: `SfC` **Siehe Lemma 3.26**

```
654 Lemma exists_R_m m : ∀ Delta, all_var_in_repo m Delta →
655     ∀ pi, { R' & R_m_aux Delta pi m = Some R'}.
```

Der Beweis für das Lemma in Listing 4.85 wird durch Induktion über `m` geführt. Der Abstraktionsfall ist erneut trivial. Für Applikationsfall kann die Hypothese darauf heruntergebrochen werden, dass sowohl die initiale Variable im Kontext ist sowie, dass die Hypothese für jeden einzelnen Term der Applikation gilt. Über die Induktionshypothese kann somit die Gesamtaussage für jeden einzelnen Term der Applikation

gezeigt werden. Es kann nun weiter gezeigt werden, dass sich diese Einzelresultate kombinieren lassen, sodass das Lemma gezeigt ist.

Grundsätzlich können die hier vorgestellten Lemmata zu der in Listing 4.86 formulierten Aussage erweitert werden.

Listing 4.86: `SfC.v`

```
724 Lemma closed_Rm : ∀ m, closed m → SfC [] (R_m_ts m) m [].
```

Es gilt sogar die umgedrehte Richtung, und eine vergleichbare Aussage für `nfty_long` (Listing 4.44). Die Beweise für beide Aussagen werden jeweils über ein Zwischenlemma geführt, das den allgemeineren Fall beweist und ähneln sich sehr stark. Dies liegt an der strukturellen Ähnlichkeit der beiden Ableitungen.

Listing 4.87: `SfC.v`

```
763 Lemma SfC_closed : ∀ R m, SfC [] R m [] → closed m.

799 Lemma Long_closed : ∀ m rho, nfty_long [] m rho → closed m.
```

4.5.6 Implementierung der typabhängigen Relation

Im Folgenden wird die typabhängige Relation R_τ formalisiert. Im Gegensatz zu R_M bedarf es hier keines Algorithmus. Wir werden an dieser Stelle die notwendige Bedingung an ein Tupelpaar formulieren, um anschließend die Liste aller gültigen Pfade zu einem Typen über diese Bedingung zu filtern. Da die `filter`-Funktion der Coq-Standardbibliothek die Filterbedingung als Funktion mit Zieltyp `bool` erwartet, ist die Bedingung als solche formuliert.

Listing 4.88: `SfC.v` **Siehe Definition 3.30**

```
850 Definition R_tau_cond (tau: type) (pipi' : path * path) : bool
    ↪  :=
851   let pi := fst pipi' in
852   let pi' := snd pipi' in
853   (pi <>b pi') &&
854     match P tau pi with
```

```
855     | None ⇒ false
856     | Some (sigma ~> tau) ⇒ false
857     | Some (? a) ⇒ match P tau pi' with
858         | None ⇒ false
859         | Some a' ⇒ (? a) ==b a'
860         end
861     end.
```

Die Menge aller gültigen Pfade in einem Typen ist genau der Definitionsbereich der Funktion P_τ, der hier der Rückgabe der Funktion `dom_P` (Listing 4.20) entspricht.

Listing 4.89: `SfC.v` **Siehe Definition 3.30**

```
864 Definition R_tau_list tau :=
865     filter (R_tau_cond tau)
866       (list_prod (dom_P tau) (dom_P tau)).
```

Auch für `R_tau_list` (Listing 4.89) benötigen wir den transitiv-symmetrischen Abschluss über `ts_cl_list` (Listing 4.53).

Listing 4.90: `SfC.v` **Siehe Definition 3.30**

```
875 Definition R_tau_ts (tau: type) := ts_cl_list (R_tau_list tau).
```

4.5.7 Zusammenhang von Langableitung und Teilformelkalkül

Als letztes Lemma wird hier Lemma 3.32, der Zusammenhang zwischen der Langableitung und des Teilformelkalküls, formalisiert. Ein wichtiger Mechanismus für den Beweis ist es, Pfadableitungen in Typableitungen zu überführen. Dafür definieren wir in Listing 4.91 zunächst eine Funktion, die Pfadkontexte in Typkontexte zu einem gegebenen Typen überführt.

Listing 4.91: `SfC.v`

```
901 Definition Delta2Gamma (tau: type) Delta : option (list type) :=
902     all_some (map (P tau) Delta).
```

Bemerkung: `all_some : list (option A) → option (list A)` ist `None`, wenn in der übergebenen Liste mindestens ein `None` vorkommt, ansonsten werden die Elemente aus dem `option` Datentyp ausgepackt und zurückgegeben.

Im Kern des Beweises für **1** ⇒ **2** (Lemma 3.32) zeigen wir, dass wir in $\vdash_R$ die Relation R durch eine Relation R' ersetzen können, wenn alle relevanten Tupel auch in R' sind. Da die Applikationsregeln bestimmen, welche Tupel relevant sind, brauchen wir nur diese Teilschritte betrachten. Dies kann als Hilfslemma formuliert werden.

Listing 4.92: `SfC.v`

```
1160 Lemma sfc_replace_R {R m Delta' pi''}:
1161     ∀ (base_sfc: SfC Delta' R m pi'') R',
1162         (∀ Delta x pi, nth_error Delta x = Some pi →
1163           ∀ ms pi' (subj_sfc: SfC Delta R (!!x @@ ms) pi'),
1164       SfC_subj _ _ _ _ _ _ _ subj_sfc base_sfc →
1165         R' (pi ++ repeat Tgt (length ms)) pi') →
1166           SfC Delta' R' m pi''.
```

Das Hilfslemma aus Listing 4.92 kann durch Induktion über `base_sfc` sowie über die anschließende Verwendung der Hypothese `(∀ Delta x pi, nth_error Delta x = Some pi → ∀ ms pi' (subj_sfc: SfC Delta R (!!x @@ ms) pi')`, bewiesen werden.

Es bleibt zu zeigen, dass, falls ein Term langtypbar ist, dieser genau diese Hypothese mit der Relation `R_tau_ts` (Listing 4.90) erfüllt. Um zu zeigen, dass dies aus den einzelnen Teilschritten einer Langableitung folgt, müssen wir zunächst die einzelnen Teilschritte dieser explizit betrachten. Dafür nutzen wir die strukturelle Ähnlichkeit zwischen der Lang- und der Pfadableitung und erstellen aus Teilschritten der Pfadableitungen Teilschritte der Langableitung. Hierfür verwenden wir die eingangs definierte Funktion `Delta2Gamma` (Listing 4.91).

Listing 4.93: `SfC.v`

```
1032 Lemma sfc_to_long_subj {rho} :
1033     ∀ Delta R m pi (base_sfc : SfC Delta R m pi) Gamma,
1034       Delta2Gamma rho Delta = Some Gamma →
```

```
1035        ∀ pr (base_long : nfty_long Gamma m (P_ok rho pi pr))
1036            Delta' m' pi' (subj_sfc : SfC Delta' R m' pi'),
1037          SfC_subj _ _ _ _ _ _ _ subj_sfc base_sfc →
1038            { Gamma' & (Delta2Gamma rho Delta' = Some Gamma') *
1039              { pr' & nfty_long Gamma' m' (P_ok rho pi' pr')}}.
```

Der Beweis für das Lemma aus Listing 4.93 folgt durch Induktion über die `SfC_subj`-Relation (Listing 4.68). Während die Regeln für die Reflexivität, Transitivität und der Abstraktion einfach folgen, muss für die Applikation eine Induktion über die Liste der Typen durchgeführt werden, die in der Langableitung mit der Funktion `make_arrow_type` (Listing 4.33) beschrieben werden.

Nun haben wir eine Möglichkeit, Schritte in Langableitungen und Pfadableitungen parallel zu betrachten. Dies erlaubt uns das in Listing 4.94 aufgestellte Lemma zu beweisen, indem der Schritt in der Langableitung `base_long` analysiert wird, der zu dem Schritt `subj_sfc` in `base_sfc` korrespondiert.

Listing 4.94: `SfC.v`

```
1099 Lemma sfc_app_subj_types_atomic {rho} :
1100   ∀ R m someDelta somepi
1101     (base_sfc : SfC someDelta R m somepi)
1102     someGamma somepr
1103     (base_long : nfty_long someGamma m (P_ok rho somepi somepr))
1104     (someD2G: Delta2Gamma rho someDelta = Some someGamma)
1105     Delta x pi (Deltaok : nth_error Delta x = Some pi)
1106     ms pi' (subj_sfc : SfC Delta R (!! x @@ ms) pi'),
1107       SfC_subj _ _ _ _ _ _ _ subj_sfc base_sfc →
1108         { a & P rho (pi ++ repeat Tgt (length ms)) = Some (? a)
              ↪ ∧
1109             P rho (pi') = Some (? a) }.
```

Mit diesem Resultat lässt sich nun das folgende Lemma beweisen, dessen Resultat die gesuchte Hypothese für `sfc_replace_R` (Listing 4.92) ist.

Listing 4.95: `SfC.v`

```
1141 Lemma sfc_app_subj_in_R_tau {rho m R} :
1142   ∀ (base_sfc : SfC [] R m []) (base_long : nfty_long [] m rho)
1143     Delta x pi (Deltaok : nth_error Delta x = Some pi) ms pi'
1144       (subj_sfc : SfC Delta R (!! x @@ ms) pi'),
1145    SfC_subj _ _ _ _ _ _ _ subj_sfc base_sfc →
1146         R_tau_ts rho (pi ++ repeat Tgt (length ms)) pi'.
```

Schließlich kann der erste Teil von Lemma 3.32 verifiziert werden.

Listing 4.96: `SfC.v` **Siehe Lemma 3.32 1 ⇒ 2**

```
1188 Lemma long_to_sfc_tau {rho m} : nfty_long [] m rho →
1189     SfC [] (R_tau_ts rho) m [].
```

Über `Long_closed` (Listing 4.87) wissen wir, dass `m` geschlossen ist. Weiter wissen wir über `closed_Rm` (Listing 4.86), dass der Term pfadableitbar mit R_M ist. Über `sfc_app_subj_in_R_tau` (Listing 4.95) und `sfc_replace_R` (Listing 4.92) wissen wir, dass wir die Relation durch `R_tau_ts rho` ersetzen können. Somit gilt das Lemma in Listing 4.96.

Die Implementierung des Schrittes **2** ⇒ **3**(Lemma 3.32) kann analog zu dem entsprechenden Schritt in Lemma 3.32 bewiesen werden.

Listing 4.97: `SfC.v` **Siehe Lemma 3.32 2 ⇒ 3**

```
1199 Lemma sfc_tau_to_Rsub_m_tau {m tau} :
1200     SfC [] (R_tau_ts tau) m [] → Rsub (R_m_ts m) (R_tau_ts
      ↪  tau).
```

Wir wissen über `R_m_ts_minimal` (Listing 4.84), dass `R_m_ts m` Teilmenge jeder Relation ist, für die es eine Pfadableitung gibt. Somit insbesondere auch für `R_tau_ts rho`. Damit gilt auch `sfc_tau_to_Rsub_m_tau` (Listing 4.97).

Es bleibt, den Schritt **3** ⇒ **1**(Lemma 3.32) zu zeigen, also dass wir aus $R_M \subseteq R_\tau$ folgern können, dass M mit τ im leeren Kontext langableitbar ist. Zunächst betrachten wir die wichtigsten Hilfslemmata, die im Beweis zu der Aussage genutzt werden.

Es kann gezeigt werden, dass, wenn ein Pfad ableitbar ist, dieser als Präfix in R verwendet wird.

Listing 4.98: SfC.v

```
1212 Lemma pi_in_R : ∀ m Delta pi R,
1213     SfC Delta R m pi → {pi' & {app & R pi' (pi ++ app)}}.
```

Des Weiteren können wir folgen, dass für gültige Pfade, die auf `Src` enden, sowohl der Pfad ohne `Src` sowie der Pfad mit `Tgt` anstelle von `Src` gültig ist, und die entsprechenden Typen in Zusammenhang stehen.

Listing 4.99: Paths.v

```
454 Lemma P_Src2 : ∀ pi rho sigma,
455     P rho (pi ++ [Src]) = Some sigma →
456       {tau & P rho pi = Some (sigma ~> tau) ∧
457         P rho (pi ++ [Tgt]) = Some tau}.
```

Wir können grundsätzlich die Aussage zeigen, dass, wenn ein Pfad in einem Typen gültig ist, auch jeder Präfix des Pfades in dem Typen gültig ist.

Listing 4.100: Paths.v

```
144 Lemma P_prefix {rho pi pi' rho'}: P rho (pi ++ pi') = Some rho'
    ↪  →
145     {rho'' & P rho pi = Some rho''}.
```

Zudem gilt, dass es zu jedem gültigen Pfad, der auf `Src` endet, einen gültigen Pfad gibt, der auf `Tgt` endet, und umgekehrt.

Listing 4.101: Paths.v

```
433 Lemma P_ok_replace_last : ∀ pi rho dir1 dir2 pr1 sigma,
434     P_ok rho (pi ++ [dir1]) pr1 = sigma →
435       {pr2 & {rho' & P_ok rho (pi ++ [dir2]) pr2 = rho'}}.
```

Zuletzt benötigen wir noch die folgende Lemmata, die einen Zusammenhang zwischen den Pfaden `pi ++ repeat Tgt n` bzw. `repeat Tgt n ++ [Src]` und dem Typen, der durch `make_arrow_type` (Listing 4.33) erstellt wird, zeigen.

Listing 4.102: `Paths.v`

```
480 Lemma P_path_make_arrow_type {tau pi n rho}:
481     P rho (pi ++ repeat Tgt n)  = Some tau →
482       {ts & P rho pi = Some (make_arrow_type ts tau) ∧
483           length ts = n}.

493 Lemma make_arrow_type_dirs {rho ts a n}:
494     make_arrow_type ts (? a) = rho →
495       P rho (repeat Tgt n ++ [Src]) = nth_error ts n.
```

Als Nächstes stellen ein Lemma auf, das den allgemeinen Fall beschreibt, dass, wenn ein Term `m` in einem beliebigen Pfadkontext zu einem beliebigen Pfad und Relation `R` pfadableitbar ist und wir aus dem Pfadkontext und einem Typen τ einen Typkontext generieren, `m` mit τ langtypbar ist, wenn `Rsub R (R_tau_ts tau)`.

Listing 4.103: `SfC.v`

```
1248 Lemma Rsub_m_tau_to_Long_aux {m tau} :
1249     ∀ Delta pi R, SfC Delta R m pi →
1250       ∀ Gamma, Delta2Gamma tau Delta = Some Gamma →
1251         Rsub R (R_tau_ts tau) →
1252           {pr & nfty_long Gamma m (P_ok tau pi pr)}.
```

Um das Lemma in Listing 4.103 zu beweisen wird zunächst eine Induktion über die Pfadableitung durchgeführt. Im Abstraktionsfall gilt es, `{pr : In pi (dom_P tau) & nfty_long Gamma (\__ s) (P_ok tau pi pr)}` zu zeigen. Zunächst beweisen wir, dass `pr` existiert. Über den Induktionsschritt wissen wir, dass der Pfad `pi ++ [Tgt]` ableitbar ist. Über `pi_in_R` (Listing 4.98) wissen wir, dass dieser Pfad in als Präfix eines Pfades `R` verwendet wird. Weiter wissen wir über die Teilmengenbeziehung auch, dass dieser in R_τ verwendet wird. Da in R_τ nur gültige Pfade vorkommen können wir schließen, dass dieser Pfad mit Präfix `pi ++ [Tgt]` ebenfalls gültig ist. Über `P_prefix` (Listing 4.100) können wir schließen, dass `pi ++ [Tgt]`, über `P_ok_replace_last`

(Listing 4.101), dass `pi ++ [Src]` und über `P_Src2` (Listing 4.99), dass `pi` ein gültiger Pfad ist, und die entsprechenden Typen zusammenhängen. Es bleibt somit zu zeigen, dass `nfty_long Gamma (\__ s) rho ~> tau'` für `P tau (pi ++ [Tgt]) = Some tau'` und `P tau (pi ++ [Src]) = Some rho` gilt. Dies lässt sich direkt über die Induktionshypothese beweisen.

Für den applikativen Fall muss gezeigt werden, dass `{pr : In pi' (dom_P tau) & nfty_long Gamma (!! x @@ ms) (P_ok tau pi' pr)}` gilt. Wir können zunächst direkt folgern, dass die Pfade `pi'` und `pi ++ repeat Tgt (length ms)`, die durch die applikative Regel in `R` enthalten sein müssen, auch in R_τ sind und somit gültige Pfade sind. Des Weiteren wissen wir über `R_tau_cond` (Listing 4.88), dass sie auf ein Typatom `? a` zeigen. Es bleibt nun zu zeigen, dass `nfty_long Gamma (!! x @@ ms) (? a)` gilt. Das Lemma `P_path_make_arrow_type` (Listing 4.102) gibt uns direkt den Beweis, dass `x` durch `make_arrow_type` (Listing 4.33) und einen atomaren Typen getypt wird, der von `NFTy_var_long` (Listing 4.44) gefordert wird. Zuletzt muss noch der Beweis erbracht werden, dass alle Terme `m` aus `ms` zu den Typen aus `make_arrow_type` passen. Dies wird mithilfe des Lemmas `make_arrow_type_dirs` (Listing 4.102) gezeigt. Der n-te Typ in der Liste von `make_arrow_type` gehört zu dem Pfad `repeat Tgt n ++ [Src]`. Nun kann mittels der Induktionshypothese der Beweis vervollständigt werden.

Wir können nun den Spezialfall von `Rsub_m_tau_to_Long_aux` (Listing 4.103) beweisen, in dem die Kontexte leer sind, was dem Fall **3** ⇒ **1**(Lemma 3.32) entspricht.

Listing 4.104: `SfC.v` — **Siehe Lemma 3.32 3 ⇒ 1**

```
1299 Lemma Rsub_m_tau_to_Long {m tau} : closed m →
1300     Rsub (R_m_ts m) (R_tau_ts tau) → nfty_long [] m tau.
```

Bemerkung: Wir benötigen hier explizit die Voraussetzung, dass `m` geschlossen ist. In Lemma 3.32 wird die Relation R_M direkt benutzt, diese existiert nach Definition 3.27 nur, wenn M auch geschlossen ist. Somit ist die Bedingung, dass M geschlossen ist, implizit durch R_M gegeben.

4.6 Abschließende Lemmata

Abschließend formalisieren wir hier Lemmata 3.33, 3.34 und 3.38 und Definition 3.36 sowie die Ersetzung aus Lemma 3.33. Diese Lemmata werden hier jedoch nur aufgestellt und nicht bewiesen.

Wir betrachten in Listing 4.105 zunächst die Ersetzung. Hierfür benötigen wir zunächst eine Formalisierung der Bedingung, dass $\pi = \pi'$, oder $(\pi, \pi') \in R_M$. Diese Bedingung können wir nun nutzen, um die Positionen in ρ zu erhalten, an denen wir ersetzen. Dies können wir durch simples Filtern erreichen. Anschließend können wir jede Position, die nicht gefiltert wurde, in ρ ersetzen.

Listing 4.105: `SfC.v`

```
1332 Definition replaceable_paths_cond m pi pi' : bool :=
1333     if (R_m_ts_dec m pi pi') then true else
1334       if (pi == pi') then true else false.

1336 Definition replaceable_paths rho m pi : list path :=
1337     filter (replaceable_paths_cond m pi) (dom_P rho).

1339 Definition replace_all_paths
1340     (rho: type)(pis : list path) (b : type) :=
1341   fold_left (replace_at_path b) pis rho.
```

Da für `replaceable_paths_cond` entschieden werden muss, ob zwei gegebene Pfade zueinander in dem transitiv-symmetrischen Abschluss von `R_m` (Listing 4.82) enthalten ist, ist es wichtig, dass dieses Enthaltensein entscheidbar ist. Die Funktion `R_m_ts_dec` entscheidet dies mittels `ts_cl_list_dec` (Listing 4.64). Siehe hierfür auch Abschnitt 4.5.2.

Wir betrachten nun eine Implementierung für die Bedingung ($\star$) aus Definition 3.36, die wichtig für die hinreichende Bedingung der prinzipale Inhabitation ist.

Listing 4.106: `Princ.v` **Siehe Definition 3.36**

```
46 Definition star tau := ∀ pi, In pi (dom_P tau) →
47     ∀ x, P tau (pi ++ [Tgt]) = Some (? x) →
48       R_tau_ts tau (pi ++ [Tgt]) (pi ++ [Tgt]).
```

Nun können die fehlenden Lemmata formuliert werden. Die Implementierung der Beweise werden jedoch in dieser Arbeit nicht behandelt. Zunächst betrachten wir Lemma 3.33, für das die Eingangs erwähnte Ersetzungsfunktion `replace_all_paths`

(Listing 4.105) geschrieben wurde. Abschließend formulieren wir die hinreichende und notwendige Bedingung der prinzipalen Inhabitation über den Teilformelkalkül.

Listing 4.107: `SfC.v` **Siehe Lemma 3.33**

```
1796 Lemma long_stays_long : ∀ m rho pi pr a, nfty_long [] m rho →
1797     P_ok rho pi pr = ? a →
1798       nfty_long [] m
1799         (replace_all_paths rho (replaceable_paths rho m pi)
1800           (fresh_type rho)).
```

Listing 4.108: `Princ.v` **Siehe Lemma 3.34**

```
106 Lemma princ_nec {m tau} : nfty_long [] m tau →
107     nflong_princ tau m → Req (R_m_ts m) (R_tau_ts tau).
```

Listing 4.109: `Princ.v` **Siehe Lemma 3.38**

```
130 Lemma princ_suff : ∀ tau, star tau → ∀ m, nfty_long [] m tau →
131     Req (R_m_ts m) (R_tau_ts tau) → nflong_princ tau m.
```

Bemerkung: `Req` bedeutet, dass `Rsub` in beide Richtungen gelten muss.

5 Fazit und Ausblick

Das direkte Resultat dieser Arbeit ist zunächst, dass Lemma 3.18 und Lemma 3.32 korrekt sind. Die hier vorgestellte Verifikation liefert zudem eine Implementierung der Teilformelfiltration und des Teilformelkalküls, die als Basis sowohl für weitere Verifikationen, sowie für weitere Forschung an der prinzipalen Inhabitation genutzt werden kann. Allen voran steht hier der Abschluss der Verifikation, der in [12] genutzten Techniken sowie dem Beantworten der dort aufgestellten Vermutung und der Implementierung des dort vorgestellten Inhabitationsalgorithmus.

5.1 Coq als Beweishelfer

Grundsätzlich lässt sich feststellen, dass Beweishelfer geeignet sind, komplexe Theorien zu verifizieren. Diese Arbeit zeigt aber auch, dass das Formalisieren von Lemmata und Implementieren von Beweisen deutlich mehr Zeit in Anspruch nimmt, als das reine mathematische Beweisen. Dies liegt mitunter daran, dass selbst offensichtliche Resultate explizit bewiesen werden müssen. Ein Beispiel hierfür ist das Lemma `P_ok_proof_irl` (Listing 4.24). Es ist intuitiv klar, dass alle Beweise für das Enthaltensein in $\mathrm{dom}(P_\tau)$ gleichwertig sind. In einer schriftlichen Beweisführung würde kein Unterschied zwischen verschiedenen Beweisen gemacht werden. In Coq muss jedoch einerseits diese Beweisirrelevanz explizit gezeigt werden, sowie dieses Resultat an allen Stellen explizit verwendet werden, in denen verschiedene Konstruktionen des Beweises für das Enthaltensein verwendet werden. Ein weiteres Beispiel sind die Generierungslemmata `SfC_gen_app` (Listing 4.72) und `SfC_gen_lam` (Listing 4.72). Diese sagen lediglich aus, dass Abstraktionen mit der Abstraktionsregel und Applikationen mit der Applikationsregel abgeleitet werden. Dies ist über die Struktur von `SfC` (Listing 4.67) offensichtlich. Für den Beweis in Coq muss jedoch ein großer Teil des Beweises als λ-Term explizit angegeben werden. Ein weiteres häufig auftretendes Problem ist, dass Induktionen in Coq nicht auf Basisfällen durchgeführt werden kann. Will beispielsweise eine Aussage über `SfC [] R m []` mit Induktion bewiesen werden, muss diese Aussage über den allgemeinen Fall `SfC Delta R m pi` getätigt werden. Somit muss in vielen Fällen

ein extra Hilfslemma erstellt werden. Siehe hierfür beispielsweise `sfc_monotone_aux` (Listing 4.73) und `Rsub_m_tau_to_Long_aux` (Listing 4.103).

Der Beweis, der den meisten Aufwand während der Implementierung in Anspruch genommen hat, ist der erste Teil des Lemmas 3.32. In schriftlicher Form können wir beschreiben, dass wir einen Beweisbaum entlang gehen und Schritte der Pfadableitung sich in Schritte der Langableitung übersetzen lassen. Um den Beweis in Coq zu Implementieren, wurden mehrere Ansätze versucht. Zunächst wurde versucht, die Aussage direkt per Induktion über die Pfadableitung zu beweisen. Nachdem dies nicht zum Erfolg geführt hat, wurde versucht einen Beweis über die Länge der Pfadableitung zu führen. Ein dritter Ansatz war es, die gesamte Aussage des Beweises in einem Datentyp zu beschreiben und die Inhabitation des Datentyps, zu zeigen. Letztendlich konnte das Lemma mit der Konstruktion `SfC_subj` (Listing 4.68) bewiesen werden. Es ist auch in der schriftlichen Beweisführung nicht unüblich, dass es mehrere Ansätze bedarf um bestimmte Aussagen zu zeigen. Die Schwierigkeit dieses Lemma zu übertragen, zeigt jedoch, dass, selbst wenn eine Aussage gut verstanden und schriftlich bewiesen wurde, es nicht direkt ersichtlich ist, wie ein solcher Beweis sich in Coq übertragen lässt.

Insbesondere im Bezug auf Coqs Standardbibliothek, trat häufiger die Situation auf, dass dort ein Lemma, das für einen Beweis benötigt wird, zwar vorhanden ist, jedoch in einer leicht anderen Form, als die, die benötigt wurde. Hier musste dann entweder die Äquivalenz zwischen der Form der Standardbibliothek und der eigenen Implementierung gezeigt werden oder das entsprechende Resultat von Grund auf selbst bewiesen werden, wie beispielsweise das Konstrukt `Forall_T`. Da die Ableitungsbeweise und Konstruktionen in `Type` formuliert wurden, konnte nicht `Forall` aus der Standardbibliothek verwendet werden, da `Forall` in `Prop` formuliert ist. `Forall_T` und `Forall` arbeiten exakt identisch und vergleichbare Aussagen über beide wurden identisch bewiesen; es musste dennoch explizit erneut aufgestellt werden. Eine Verallgemeinerung des Datentyps `Forall` zu `Type` wurde dieses Problem lösen. Dies wäre auch ganz im Sinne der Prinzipalität.

In schriftlichen Beweisen ist es zudem einfacher, bereits bewiesene Resultate in externen Arbeiten zu zitieren. In dem Fall der Enscheidbarkeit des transitiven Abschlusses (Siehe Abschnitt 4.5.2) könnte auf die Bibliothek math-comp[1] verwiesen werden, in der dies über eine Tiefensuche bewiesen wurde. Da dieser Beweis eine so andere Implementierung von Graphen nutzt, ist es einfacher gewesen die Entscheidbarkeit selbst zu zeigen, als eine Äquivalenz der beiden Implementierungen.

[1] https://github.com/math-comp/math-comp

Grundsätzlich sorgen diese Eigenschaften für deutlich mehr Arbeitsaufwand, als das Beweisen auf Papier, jedoch führen sie zu einer Form, die der Typchecker von Coq einfach Verifizieren kann. Insgesamt wurde für die Implementierung 5631 Zeilen Code geschrieben[2].

Es ist zu erwarten, dass die genannten Probleme auch für weitergehende Verifikationen im λ-Kalkül gelten, jedoch ist ebenfalls anzunehmen, dass hierdurch eine umfassende Bibliothek an Lemmata entsteht, die das Formulieren und Beweisen von Aussagen im λ-Kalkül kontinuierlich vereinfachen. An dieser Stelle seinen noch einige Ideen zur Verbesserung der Implementierung gegeben. Zum einen wuchs das Wissen um Coq kontinuierlich mit Fortschritt der Implementierung, sodass Lemmata, die zu Beginn aufgestellt wurden unstrukturierter und schwieriger nachzuvollziehen sind, als jene, die in am Ende aufgestellt und bewiesen wurden. Hier kann im Bezug auf die Lesbarkeit einiges verbessert werden. Zum anderen wurde wenige eigene Taktiken mittels L_{tac}[3] umgesetzt auch hierdurch kann die Lesbarkeit noch erhöht werden. Zuletzt wurde nur wenig Gebrauch von *hint databases* gemacht, die die Taktik `auto` befüllen.

5.2 Ausblick

Die prinzipale Inhabitation und die Prinzipalität an sich, sind noch verhältnismäßig wenig erforschte Themen. Eine grundlegende Arbeit ist hier [6] von 1999, in der ein Algorithmus vorgestellt wird, um prinzipale Inhabitanten eines Typs zu zählen. In neuerer Zeit ist die prinzipale Inhabitation Thema von [12], sowie von [1]. Letzteres erweitert die in [18] aufgestellten *Inhabitationsautomaten*, um Resultate über die prinzipale Inhabitation zu erhalten. In [12] wird vorgeschlagen, die prinzipale Inhabitation auch in anderen Typsystemen zu untersuchen. Hier werden allen voran die Intersektionstypen [9] genannt.

Interessant ist auch die Untersuchung in konkreten Programmiersprachen. Betrachten wir das in der Einleitung erwähnte Beispiel der Identität. Eine Compiler könnte feststellen, dass der Typ `int` $\to$ `int` nicht der prinzipale Typ der Identität ist, und dies mit einer Warnung anmerken und einen prinzipalen Typ generieren. Hierdurch kann mehrfacher Code vermieden werden. Tatsächlich bietet OCaml die Option `-principal`, die die eingebaute Typableitung so gestaltet, dass alle Typen prinzipal sind.[4]

[2] Gezählt durch `cloc`

[3] `https://coq.inria.fr/refman/proof-engine/ltac.html`

[4] Siehe `man 1 ocaml` oder `https://linux.die.net/man/1/ocaml`

Über den *Curry-Howard-Isomorphismus* kann untersucht werden, welche Bedeutung eine prinzipale Aussage zu einem Beweis hat. Auf den ersten Blick entspricht dies einem Beweis, der nicht mehr beweist, als die Aussage, für die er aufgestellt wurde. Hiermit könnten allgemeinere Resultate zu bestehenden Beweisen gefunden werden. Im Bezug auf die Inhabitation könnte dies zu einer stärkeren Form der *Beweisnormalisierung*[19] führen, sodass nicht nur normalisierte Beweise generiert werden, sondern insbesondere *kleinste* normalisierte Beweise.

Die Erweiterung des *Curry-Howard-Isomorphismus* zum *Curry-Howard-Lambek-Isomorphismus*[5] erlaubt es, die Prinzipalität in den kartesisch abgeschlossenen Kategorien zu untersuchen. Grundsätzlich sagt diese Erweiterung aus, dass Typen Objekte einer kartesisch abgeschlossenen Kategorie sind, deren Morphismen die vom Zielobjekt inhabitierte λ-Terme mit einer freien Variable des Quellobjektes sind. Eine kartesisch abgeschlossene Kategorie erfordert unter anderem, dass zu jeden Objekten A und B sowohl das Produkt $A \times B$ als auch die Potenz A^B existieren. Letzteres können wir auch als $B \to A$ schreiben und entspricht somit einem Pfeiltypen. Hier kann untersucht werden, ob die Einschränkung auf prinzipale Inhabitanten ebenfalls eine Kategorie ergibt, welche Eigenschaften diese hat, und inwieweit dies dem Verständnis der Prinzipalität dient.

Wir schließen mit einer letzten Betrachtungsweise prinzipaler Typen, indem wir Definition 3.1 und Lemma 3.2 betrachten. Die Subtyprelation $\preceq$ zusammen mit $\mathbb{T}_M$, der Menge aller Typen, die von einem gegebenen Term M inhabitiert werden, ergeben den Verband $(\mathbb{T}_M, \preceq)$, dessen Minimum genau der prinzipale Typ von M ist. Diese Beobachtung ist eng verwandt mit dem allgemeineren Konzept der *prinzipalen Algebra*[3], in denen der prinzipale Typ final ist.

Wir können also schließen, dass sowohl die Prinzipalität und prinzipale Inhabitation als auch die Verifikation mittels Coq eine wichtige Rolle in zukünftiger Forschung spielen werden.

Literatur

[1] Sandra Alves und Sabine Broda. „Inhabitation machines: determinism and principality“. In: *NCMA*. Österreichische Computer Gesellschaft, 2017, S. 57–70.

[2] Nada Amin und Ross Tate. „Java and Scala's Type Systems Are Unsound: The Existential Crisis of Null Pointers“. In: *SIGPLAN Not.* 51.10 (Okt. 2016), S. 838–848. ISSN: 0362-1340. DOI: 10.1145/3022671.2984004.

[3] Henk Barendregt, Wil Dekkers und Richard Statman. *Lambda Calculus with Types*. New York, NY, USA: Cambridge University Press, 2013. ISBN: 9780521766142.

[4] Jan Bessai u. a. „Combinatory Logic Synthesizer“. In: *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*. Hrsg. von Tiziana Margaria und Bernhard Steffen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, S. 26–40. ISBN: 978-3-662-45234-9. DOI: 10.1007/978-3-662-45234-9_3.

[5] G. M. Bierman und V. C. V. de Paiva. „On an Intuitionistic Modal Logic“. In: *Studia Logica* 65.3 (Aug. 2000), S. 383–416. ISSN: 1572-8730. DOI: 10.1023/A:1005291931660.

[6] Sabine Broda und Luís Damas. „Counting a Type's Principal Inhabitants“. In: *Typed Lambda Calculi and Applications*. Hrsg. von Jean-Yves Girard. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, S. 69–82. ISBN: 978-3-540-48959-7. DOI: 10.1007/3-540-48959-2_7.

[7] Ashok K. Chandra, Dexter C. Kozen und Larry J. Stockmeyer. „Alternation“. In: *J. ACM* 28.1 (Jan. 1981), S. 114–133. ISSN: 0004-5411. DOI: 10.1145/322234.322243.

[8] B. Jack Copeland. „The Church-Turing Thesis“. In: *The Stanford Encyclopedia of Philosophy*. Hrsg. von Edward N. Zalta. Winter 2017. Metaphysics Research Lab, Stanford University, 2017. URL: https://plato.stanford.edu/archives/win2017/entries/church-turing/.

[9] M. Coppo und M. Dezani-Ciancaglini. „An extension of the basic functionality theory for the λ-calculus.“ In: *Notre Dame J. Formal Logic* 21.4 (Okt. 1980), S. 685–693. DOI: 10.1305/ndjfl/1093883253.

[10] Thierry Coquand. „An Analysis of Girard's Paradox“. In: *In Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1986, S. 227–236. DOI: 10.1.1.37.3153.

[11] Haskell B. Curry. „On the Definition of Substitution, Replacement and Allied Notions in a Abstract Formal System“. In: *Revue Philosophique De Louvain* 50.26 (1952), S. 251–269.

[12] Andrej Dudenhefner und Jakob Rehof. „The Complexity of Principal Inhabitation“. In: *2nd International Conference on Formal Structures for Computation and Deduction (FSCD 2018)*. Hrsg. von Dale Miller. Bd. 84. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 15:1–15:14. ISBN: 978-3-95977-047-7. DOI: 10.4230/LIPIcs.FSCD.2017.15. URL: http://drops.dagstuhl.de/opus/volltexte/2017/7735.

[13] Andrej Dudenhefner und Jakob Rehof. „Typability in bounded dimension“. In: *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. Juni 2017, S. 1–12. DOI: 10.1109/LICS.2017.8005127.

[14] J. Roger Hindley. *Basic Simple Type Theory*. Bd. 42. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2008. ISBN: 9780521054225. DOI: 10.1017/CBO9780511608865.

[15] Fairouz Kamareddine. „Reviewing the Classical and the de Bruijn Notation for λ-calculus and Pure Type Systems“. In: *Journal of Logic and Computation* 11.3 (2001), S. 363–394. DOI: 10.1093/logcom/11.3.363.

[16] Steven Schäfer, Tobias Tebbi und Gert Smolka. „Autosubst: Reasoning with de Bruijn Terms and Parallel Substitutions“. In: *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015*. Hrsg. von Xingyuan Zhang und Christian Urban. LNAI. Springer-Verlag, Aug. 2015. DOI: 10.1007/978-3-319-22102-1_24.

[17] Moses Schönfinkel. „Über die Bausteine der mathematischen Logik“. In: *Mathematische Annalen* 92.3 (Sep. 1924), S. 305–316. ISSN: 1432-1807. DOI: 10.1007/BF01448013.

[18] Aleksy Schubert, Wil Dekkers und Henk P. Barendregt. „Automata Theoretic Account of Proof Search“. In: *24th EACSL Annual Conference on Computer Science Logic (CSL 2015)*. Hrsg. von Stephan Kreutzer. Bd. 41. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, S. 128–143. ISBN: 978-3-939897-90-3. DOI: 10.4230/LIPIcs.CSL.2015.128. URL: http://drops.dagstuhl.de/opus/volltexte/2015/5411.

[19] Morten Heine Sørensen und Paweł Urzyczyn. *Lectures on the Curry-Howard Isomorphism*. Bd. 149. Studies in Logic and the Foundations of Mathematics. New York, NY, USA: Elsevier Science Inc., 2006. ISBN: 0444520775.

[20] A. S. Troelstra und H. Schwichtenberg. *Basic Proof Theory*. New York, NY, USA: Cambridge University Press, 1996. ISBN: 0-521-57223-1.

[21] Paweł Urzyczyn. „Inhabitation in typed lambda-calculi (a syntactic approach)“. In: *Typed Lambda Calculi and Applications*. Hrsg. von Philippe de Groote und J. Roger Hindley. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, S. 373–389. ISBN: 978-3-540-68438-1. DOI: 10.1007/3-540-62688-3_47.

Eidesstattliche Versicherung

Stahl, Christoph	**116531**
Name, Vorname	Matr.-Nr.

Ich versichere hiermit an Eides statt, dass ich die vorliegende Masterarbeit mit dem Titel

Prinzipale Inhabitation im einfach getypten Lambda-Kalkül

selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Ort, Datum	Unterschrift

Belehrung:
Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -)
Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.
Die Technische Universität Dortmund wird gfls. elektronische Vergleichswerkzeuge (wie z.B. die Software *turnitin*) zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.
Die oben stehende Belehrung habe ich zur Kenntnis genommen:

Ort, Datum	Unterschrift