Indizierte Typen

Zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften
von der Fakultät für Informatik
der Universität Karlsruhe (Technische Hochschule)
genehmigte

Dissertation

von Christoph Zenger
aus München

Tag der mündlichen Prüfung: 17. Juli 1998
Erster Gutachter: Prof. Dr. Jacques Calmet
Zweiter Gutachter: Priv.-Doz. Dr. Peter Thiemann

Inhaltsverzeichnis

Danksagung

Ich möchte mich an dieser Stelle bei all denen bedanken, die zum Gelingen dieser Arbeit beigetragen haben:

Mein Betreuer, Prof. Jacques Calmet, hat mir ein Themengebiet nahegebracht, das mich immer noch fasziniert. Er hat mir ein ungewöhnliches Maß an Freiheit zugestanden, mich immer unterstützt und auch persönlich Anteil genommen, als mir die Arbeit mal schwerer fiel. Er hat mir die Möglichkeit eröffnet, an vielen Konferenzen und Workshops teilzunehmen.

Dr. Peter Thiemann hat das zweite Gutachten übernommen. Er hat sich über das übliche Maß hinaus mit meiner Arbeit auseinandergesetzt und durch seine gründliche Korrektur und detaillierten Verbesserungsvorschläge hat die Arbeit viel hinzugewonnen.

Meine Kollegen Werner Seiler, Winfried Fakler, Joachim Schue, Karsten Homann und Peter Kullmann waren eine Arbeitsumgebung, in der ich mich wohlgefühlt habe. Besonders Werner war, obwohl fachfremd, immer bereit, meine Probleme zu diskutieren.

Mit meinen Informatikerfreunden Martin Wehr, Matthias John und Wolfgang Ahrendt konnte ich unzählige Diskussionen führen. Ihre Begeisterung und ihr Interesse haben mich immer wieder angesteckt und motiviert.

Viele weitere Freundinnen und Freunde, Birte, Brigitte, Chrischan, Karsten, Kirsten und Lorenz aus der Wohngemeinschaft und Astrid, Christine, Gerhard, Joachim und Stephen aus der Meisenthal-Clique haben das Leben in dieser Zeit mit mir geteilt und mir in Karlsruhe ein zuhause gegeben.

Meine Familie war mir ein sicherer Rückhalt. Die Diskussionen mit meinem Vater waren immer wieder eine Inspiration.

Karsten und Matthias haben die Arbeit gründlich korrekturgelesen.

Vielen Dank Euch allen.

Kapitel 1

Einführung

In dieser Arbeit wird ein neues Typsystem für eine funktionale Programmiersprache vorgestellt. Dieses Typsystem erlaubt die Verwendung *indizierter Typen*. Indizierte Typen sind Typen, die mit bestimmten Werten wie ganzen Zahlen oder Wahrheitswerten parametrisiert werden können.

Die Motivation war es, ein Typsystem zu entwerfen, das die Kompatibilität von Matrizen bei der Matrixmultiplikation *statisch*, also zur Übersetzungszeit, sicherstellen kann. Das entstandene System ist darüber hinaus in der Lage, die meisten Reihungsgrenzen und einige Invarianten von Datenstrukturen, beispielsweise die Balance-Eigenschaft von AVL-Bäumen, statisch zu überprüfen.

In manchen Fällen, wenn zum Beispiel die Größe der Matrizen erst zur Laufzeit eingelesen wird, kann die Korrektheit prinzipiell erst zur Laufzeit überprüft werden. Das Typsystem erlaubt es dann, *dynamische* Überprüfungen einzubauen. Trotz der dynamischen Tests ist statisch garantiert, daß zur Laufzeit nur typkorrekte Ausdrücke ausgewertet werden.

Die Arbeit schließt die Entwicklung der Typinferenz, eines neuen, allgemeineren Unifikationsalgorithmus, deren Korrektheits- und Vollständigkeitsbeweise und eine Prototypimplementierung ein.

1.1 Typen in Computersprachen

Computersprachen spielen in der Informatik eine wesentliche Rolle. In Computersprachen formulieren wir letztlich die Software, die wir erstellen. Aber darüber hinaus prägen sie auch die Art, in der wir über Software-Lösungen

nachdenken, da wir Lösungen oft mit den Strukturierungsmechanismen, die uns unsere Programmiersprache bietet, beschreiben.

Typen werden dazu benutzt, Eigenschaften von Programmteilen, bzw. deren Werte zur Laufzeit, zu beschreiben. Sie hängen also eng mit den Strukturierungsmechanismen der Programmiersprache zusammen. Beim Zusammensetzen von Programmkomponenten stellt der Übersetzer dann neben der syntaktischen Kompatibilität – in der Regel wird eine kontextfreie Grammatik angegeben, der die Programme genügen müssen – auch noch die Typkorrektheit sicher. Dies erlaubt, Aussagen über das Laufzeitverhalten zu machen, und bestimmte Arten von Fehlern zur Laufzeit auszuschließen.

1.2 Vorteile von Typen

Was spricht dafür, Typen in eine Computersprache einzuführen? Die zumeist genannten Gründe sind

- Fehlererkennung beim Übersetzen,
- Sicherheitsaussagen über erzeugte Programme,
- Effizienz übersetzter Programme,
- Dokumentation und
- Interoperabilität.

Die frühzeitige Fehlererkennung ist oft der wichtigste Grund. Es gibt viele Fehler, die von der Typüberprüfung bereits zur Übersetzungszeit erkannt werden können. Dazu gehören eine falsche Reihenfolge der Operanden, wenn sie einen unterschiedlichen Typ haben, das Verwechseln unterschiedlicher, aber ähnlicher Strukturen (Dateizeiger und Dateideskriptor) und viele mehr. Je differenzierter wir die Laufzeitgrößen mit Typen unserer Programmiersprache charakterisieren, umso mehr Fehler können von der Typüberprüfung gefunden werden. Diese Fehlererkennung spielt bei Spezifikationssprachen, bei denen wir keine Tests durchführen können, eine besonders wichtige Rolle. Das in der Arbeit entwickelte Typsystem legt auf diese frühzeitige Fehlererkennung den Schwerpunkt, indem es dem Programmierer die Möglichkeit gibt, bei den Typen mehr als gewöhnlich zu differenzieren und zum Beispiel Matrizen verschiedener Größe verschiedene Typen zuzuordnen.

Fehler, die bereits zur Übersetzungszeit gefunden werden, können zur Laufzeit nicht mehr auftreten. Aber die Sicherheitsaussagen, die wir über typgeprüfte Programme machen können, gehen darüber hinaus. Es werden nicht nur bestimmte Fehlerklassen aus Sicht des Programmierers entdeckt, sondern es können auch Fehlerklassen aus Sicht des Endbenutzers ausgeschlossen werden. In vielen typisierten Sprachen wie Standard ML [MTH90], Haskell [PHA+97] oder Java [GJS96], sind „core dumps“ zur Laufzeit ausgeschlossen, vorausgesetzt der Übersetzer ist korrekt.

Besonders deutlich wird dieser Aspekt von Typen bei manchen Beweisern. Dort stehen nicht nur gewisse Eigenschaften von Argument und Resultat im Typ einer Funktion, sondern alle, für die Korrektheit der Funktion relevanten, Aussagen. Diese müssen dann durch die Funktion „bewiesen“ werden. Die Typüberprüfung überprüft nun sogar die Korrektheit der Funktion.

Die Typinformation, die der Übersetzer aus Typinferenz oder Typdeklarationen erhält, kann er auch benutzen um effizienteren Zielkode zu erzeugen. In Smalltalk [GR89] muß eine Additionsroutine zur Laufzeit untersuchen, welche Klasse die Operanden haben, in C [KR78] wird die richtige Operation zur Übersetzungszeit herausgefunden. In dieser Arbeit trifft dieses Argument vor allem auf die Überprüfung der Reihungsgrenzen zu. Wenn die Zugriffe statisch als korrekt erkannt werden, kann die Überprüfung zur Laufzeit wegfallen.

Auch dem Programmierer als Leser – die meisten Programme werden sehr viel öfter gelesen als geschrieben – geben, neben dem Namen einer Funktion, die Typen von Argumenten und Resultat einen wesentlichen Hinweis darauf, was eine Funktion macht. Als Beispiel seien hier die Funktionsdeklarationen

```
intersect: set x set -> set
intersect: rectangle x rectangle -> rectangle
```

gegeben. In beiden Fällen ist durch den Typ klar, was die Funktion `intersect` macht. Ohne die Typdeklaration wäre das nicht der Fall. Insbesonders bei Schnittstellen größerer Softwarepakete oder Bibliotheken ist dieser Dokumentationseffekt von Typen nicht zu unterschätzen. Prechelt und Tichy zeigen in einer Studie [PT98], daß Typen meßbare Erfolge bringen.

Bei der Interoperation zwischen verschiedenen Softwarepaketen wird oft angenommen, daß gleiche Namen für gleiche Dinge stehen. Diese Annahme kann manchmal unzutreffend sein und birgt daher immer ein Risiko. Keine sichere, aber wesentlich zuverlässigere Annahme ist es, darauf zu vertrauen,

daß gleiche Namen, für die der selbe Typ angegeben ist, die selbe Sache bezeichnen [WWRT91].

1.3 Nachteile von Typen

Warum gibt es dann Sprachen, die nicht typisiert sind? Es gibt im wesentlichen drei Gründe:

- Flexibilität,
- Reflektion und
- mühevolle Notation.

Betrachten wir einen Algorithmus zur Exponentiation. In C müssen wir diesen Algorithmus mehrmals aufschreiben, einmal für `double`, einmal für `int` — und später für `(struct polynomial *)` noch einmal. In einer untypisierten Sprache wie Smalltalk haben wir dieses Problem nicht, wir notieren den Algorithmus einmal. Das doppelte Schreiben von Funktionen ist zusätzlicher Aufwand und, insbesonders beim Verändern von Programmen, fehleranfällig, da wir alle Versionen parallel ändern müssen. Es gibt Typsysteme mit Überladung, die das Beispiel der Exponentiation besser behandeln als C, aber kein effektives Typsystem kann alle sinnvollen Programme, also alle, bei denen keine Laufzeitfehler auftreten, typisieren.

In manchen Programmiersprachen wie Scheme [SS78] ist ausführbarer Kode eine Datenstruktur wie jede andere und kann zur Laufzeit erzeugt und ausgeführt werden. Es ist keine Trennung von Laufzeit und Übersetzungszeit mehr möglich. Beispielsweise bei partieller Evaluierung [JGS93] ist diese Möglichkeit wichtig. Inzwischen gibt es auch hier Ansätze, Typen zu verwenden [Hug96], aber diese sind komplizierter als ungetypte Ansätze.

Je flexibler ein Typsystem ist, umso mehr Typannotationen sind notwendig. Wenn Typen Funktionen sehr differenziert beschreiben, können die Typen oft groß werden [PL89], und Programme mit vielen Typannotationen sind dann manchmal schwer zu lesen.

Bei Skriptsprachen wie der UNIX shell laufen viele Programme nur wenige Male. Meist wird dann auf ein Typsystem verzichtet, um dem Programmierer optimale Flexibilität, Reflektion und knappe Notation zu erlauben.

1.4 Statisch oder dynamisch?

Man unterscheidet statische und dynamische Typüberprüfung. Bei einem statischen Typsystem werden alle Überprüfungen zur Übersetzungszeit gemacht, bei dynamischen Typsystemen [Hen93a] werden manche Typfehler erst zur Laufzeit gefunden.

Dynamische Typüberprüfung kann mehr Konstruktionen zulassen, aber insbesondere bei Frameworks oder Bibliotheken, die eventuell nicht einmal im Quellcode ausgeliefert werden, ist ein Typfehler zur Laufzeit besonders störend.

Der Ansatz dieser Arbeit ist es, dem Programmierer die Möglichkeit einer dynamischen Typüberprüfung zu geben, aber dennoch statisch Typsicherheit zu gewährleisten. Die Technik ist eine Art spezielles `if` für Typüberprüfungen, wobei im `then` Teil der erfolgreiche Ausgang der Typüberprüfung statisch zur weiteren Typüberprüfung verwendet werden kann. Dies hat gegenüber übersetzererzeugten, dynamischen Überprüfungen den Vorteil, daß dynamische Typüberprüfungen vom Programmierer bewußt eingesetzt werden müssen.

1.5 Motivation für indizierte Typen

Die Motivation für diese Arbeit kam aus folgendem Beispiel einer Matrixmultiplikation, die ein inkorrektes Ergebnis liefert (die Dimensionen passen nicht).

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} = \begin{pmatrix} 9 & 12 & 15 \\ 19 & 26 & 33 \\ 29 & 40 & 51 \end{pmatrix}$$

Hier ist die korrekte Multiplikation mit dem korrekten Ergebnis:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} = \begin{pmatrix} 22 & 28 \\ 49 & 64 \\ 76 & 100 \end{pmatrix}$$

Die Berechnung des ersten Ergebnisses beruht auf dem dubiosen Skalarprodukt

$$\begin{pmatrix} 1 \\ 2 \end{pmatrix} \begin{pmatrix} 1 \\ 4 \\ 7 \end{pmatrix} = 9,$$

das von folgender Implementierung des Skalarprodukts in Haskell als Ergebnis geliefert wird.

```
sprod x y = (sum (zipWith * x y))
```

Erschreckend ist an diesem Beispiel besonders, daß nicht einmal zur Laufzeit eine Fehlermeldung auftritt. Bei geeigneter Programmierung würde zwar zur Laufzeit eine Fehlermeldung erzeugt, aber zur Übersetzungszeit kann ein solcher Fehler in Haskell nicht gefunden werden.

Wir wollen erreichen, daß diese Art von Fehler bereits bei der Typüberprüfung ausgeschlossen wird, indem wir den Typ einer Matrix mit der Größe parametrisieren und die Matrixmultiplikation entsprechen deklarieren:

```
mat_mult :: (Matrix Int m n) -> (Matrix Int n k)
         -> (Matrix Int m k);
```

Das erste Beispiel führt nun zu einem Typfehler. Diese Überprüfung scheint einfach, weil hier die Größen der Matrizen Konstanten sind. Die Typüberprüfung muß aber auch verifizieren, daß `mat_mult` die Routine für das Skalarprodukt

```
s_prod :: (Vector Int n) -> (Vector Int n) -> Int;
```

nur mit passenden Typen aufruft. Hierbei müssen alle möglichen korrekten Aufrufe von `mat_mult` in Betracht gezogen werden.

Neben der linearen Algebra können wir noch weitere interessante Probleme mit indizierten Typen behandeln, die statische Überprüfung von Reihungsgrenzen und die statische Überprüfung bestimmter Invarianten von Datenstrukturen, wie die Balance-Bedingung bei AVL-Bäumen.

Die hier entwickelte Methode kann auch die Grundlage bilden für weitere Typsysteme, die es erlauben, Typen mit allgemeineren Werten zu parametrisieren.

1.6 Probleme

Hier soll ein kurzer Überblick über die technischen Schwierigkeiten gegeben werden, die bei der Typinferenz für indizierte Typen entstehen. Es ist offensichtlich, daß das Typinferenzproblem komplizierter wird, da Bedingungen an die Indizes zu den Bedingungen an die Typen hinzukommen. Es entstehen aber noch weitere Schwierigkeiten.

Wir betrachten einen einfachen indizierten Typ, den Typ Vektor. Der Unterschied zum Typ Liste, wie er auch bei anderen funktionalen Sprachen üblich ist, besteht in dem zusätzlichen Längenparameter.

```
data List a = Cons a (List a)
            | Nil;
data Vector a #n = Vcons a (Vector a (n - 1)) <= (n > 0)
                 | Vnil <= (n = 0);
```

Als Beispiel wollen wir zuerst eine Längenfunktion auf Listen mit einer entsprechenden auf Vektoren vergleichen.

```
len :: (List a) -> Int;
len (Nil) = 0;
len (Cons x xs) = (len xs) + 1;

vlen :: (Vector a n) -> Int;
vlen (Vnil) = 0;
vlen (Vcons x xs) = (vlen xs) + 1;
```

Im Muster der Funktion `len` haben `(Nil)` und `(Cons x xs)` den allgemeinen Argumenttyp `(List a)`. In der Tat würde mit einer weiteren Definitionsgleichung mit dem speziellen Argumenttyp `(List Int)`

```
len [1] = 1;
```

`len` nur noch den Typ `(List Int) -> Int` haben. Es könnten ja sonst bei einem Aufruf `(len (Cons True Nil))` der Wert `True` und der Wert `1` verglichen werden, obwohl diese verschiedenen Typ haben.

Wenn wir nun aber `vlen` betrachten, sehen wir, daß das erste Muster `(Vnil)` den Typ `(Vector a 0)`, also nicht den allgemeinen Typ hat. Die in dieser Arbeit vorgeschlagene Lösung unterscheidet Gleichheit von Typen und *strukturelle* Gleichheit von Typen, was soviel heißt wie Gleichheit unter Nichtbeachtung von Indizes. Die Argumente müssen dann nur von strukturell allgemeinem Typ sein. Bei der Musteranpassung werden dann Vektoren verschiedener Länge verglichen, aber es kann nie `1` mit `True` verglichen werden.

Für das nächste Problem betrachten wir die `vmap`-Funktion, das Analogon zur `map`-Funktion auf Listen:

```
vmap :: (a -> b) -> (Vector a n) -> (Vector b n);
vmap f (Vnil) = (Vnil);
vmap f (Vcons x xs) = (Vcons (f x) (vmap f xs));
```

Als Ergebnis im ersten Zweig von `vmap` hat `(Vnil)` den Typ `(Vector Int 0)`. Wir können aber leicht sehen, daß dieser Zweig nur dann relevant ist, wenn das zweite Argument `(Vnil)` ist, also auch den Typ `(Vector Int 0)` hat. Dann ist aber $n = 0$ und somit ist das Resultat immer von geeignetem Typ. Eine ähnliche, etwas kompliziertere Schlußfolgerung führt auch beim zweiten Zweig zum Erfolg. Wir müssen also Fakten, die wir beim Musteranpassen über die Indizes erfahren, in der Typinferenz verwenden. Dies führt dazu, daß die case-Regel eine nicht-triviale Erweiterung des Hindley-Milner-Systems darstellt und daß die Unifikation auch Implikationen zwischen Bedingungen behandeln muß.

1.7 Resultate

Was bietet die Arbeit nun Neues? Zum Ersten beschreiben wir ein neues Typsystem für eine funktionale Programmiersprache, das es erlaubt, Datentypen mit Werten wie ganzen Zahlen und Wahrheitswerten zu parametrisieren. Dies eröffnet gegenüber bisherigen Typsystemen neue Möglichkeiten. Wir können die Dimensionen in der linearen Algebra, viele Reihungsgrenzen und Invarianten von Datenstrukturen statisch überprüfen. Allgemeiner können wir viele Probleme, die zuvor nur mit abhängigen Typen bearbeitet werden konnten, nun in einer Programmiersprache behandeln, die den Standardsprachen des funktionalen Programmierens sehr ähnlich ist.

Technisch führen wir dafür einige Neuerungen ein. Die Aufteilung zwischen Typinferenz und Unifikation ist zwar nicht eigentlich neu, ist aber in dieser Arbeit strikter und expliziter als in der Literatur üblich. Weiter führt diese Arbeit eine Unterscheidung zwischen *strukturell* gleichen Typen und gleichen Typen ein, die die Behandlung indizierter Typen erst ermöglicht. Für die case-Regel wird eine Behandlung mit Implikation vorgeschlagen, die auch für andere Typsysteme relevant sein kann. Dies erfordert einen neuen Algorithmus für die Unifikation. Auch die Beweise für Korrektheit, Vollständigkeit und Subjektreduktion insbesondere der case-Regel, sind nicht-triviale Erweiterungen.

1.8 Überblick

Wir werden im nächsten Kapitel einen grundlegenden Überblick über Typsysteme und Polymorphie geben. Daran anschließend werden wir Indexsysteme einführen, wie wir sie für eine Sprache mit indizierten Typen brauchen. Das vierte Kapitel ist der eigentliche Kern der Arbeit. Hier wird das Typsystem vorgestellt, es werden Inferenz und Unifikation erläutert, Aussagen zur operationellen Korrektheit des Systems gemacht und die Komplexität der Typinferenz besprochen. Im fünften Kapitel werden wir die praktische Anwendung von indizierten Typen an Beispielen diskutieren und die Grenzen des Systems erörtern. Hier findet sich auch ein Vergleich mit abhängigen Typen. Am Ende fassen wir die Arbeit nochmals zusammen und geben einen Ausblick über mögliche Weiterentwicklungen. Die Beweise sind im Anhang zusammengefaßt.

Kapitel 2

Typsysteme

In diesem Kapitel wollen wir die Grundlagen von Typsystemen darlegen. Wir orientieren uns dabei an Barendregt [Bar92]. Eine weitere grundlegende Darstellung findet sich bei Mitchell [Mit90]. Andere Bücher betrachten mehr den semantischen Aspekt von Typen [Mit96], [Gun92], [Cro93].

Ein Typsystem unterteilt Programme in *korrekt getypte* Programme und *nicht korrekt getypte*. Was wir aber eigentlich wollen, ist eine Unterteilung in *sinnvolle* und *unsinnige* Programme. Bei einer Funktionsanwendung sollte die Funktion wirklich eine Funktion sein und das Argument aus ihrem Definitionsbereich. Wenn solche oder ähnliche Fehler während der Ausführung des Programms auftreten können, bezeichnen wir es als unsinnig, andernfalls als sinnvoll. Ziel ist es nun, Typsysteme zu finden, die korrekt sind, die also unsinnige Programme als nicht typkorrekt ablehnen, andererseits aber möglichst viele sinnvolle Programme akzeptieren. Ein korrektes und vollständiges Typsystem, in dem alle sinnvollen Programme akzeptiert werden, und das gleichzeitig überprüfbar ist, kann es nicht geben. Der Ausdruck [Jon94]

```
1 + (if True then 1 else "xyz")
```

wird von vielen typisierten Sprachen abgelehnt, da die Typüberprüfung nicht vorhersagen kann, daß immer der erste Zweig gewählt wird. Weiterentwickelte Typsysteme würden vielleicht obigen Ausdruck akzeptieren, aber bei komplizierteren Ausdrücken für `True` scheitern.

Wir fangen dieses Kapitel mit einem Abschnitt über den ungetypten Lambdakalkül an, der den betrachteten funktionalen Programmiersprachen als Berechnungsmodell zugrunde liegt [Bar84]. Wir werden dann typisierte

Lambdakalküle ansehen, solche mit sehr einfachen, aber auch solche mit sehr flexiblen Typsystemen.

Typen können auch als Mittel zur Spezifikation eingesetzt werden. Der Typ einer Funktion entspricht dann ihrer Spezifikation und die Definition der Funktion enthält auch ihren Korrektheitsbeweis. Hier sind sinnvolle Programme gewissermaßen nur solche, die der Spezifikation entsprechen. Beispiele für Beweiser, die dies ausnutzen sind NuPrl [CAB+86], Coq [CCF+95] und LEGO [LP92].

Am Ende des Kapitels wollen wir die Typrekonstruktion genauer betrachten. Diese ist in einer Programmiersprache notwendig, da die Annotation aller Typen die Lesbarkeit eines Programms stark beeinträchtigen würde. Wir sehen uns den Typrekonstruktionsalgorithmus von Milner [Mil78] ausführlicher an. Auf ihm bauen die Typrekonstruktionsalgorithmen für funktionale Sprachen auf. Wir geben auch noch eine zweite Beschreibung des Typrekonstruktionsalgorithmus, der die Typinferenz in zwei Phasen unterteilt, die eigentliche Inferenz und die Unifikation. Eine solche Unterteilung findet sich auch bei Wand [Wan87b]. Diese Unterteilung hat den Vorteil, daß die beiden Probleme getrennt behandelt werden können, was insbesonders Erweiterungen des Systems vereinfacht. Wir werden einige Erweiterungen ansprechen und wollen dann im weiteren Verlauf der Arbeit indizierte Typen entwickeln, eine Programmiersprachenvariante für abhängige Typen.

2.1 Der ungetypte Lambdakalkül

Wenn wir in der Mathematik eine Funktion definieren, unterscheiden wir Argumente von Parametern, indem wir die Argumente explizit angeben. Wir schreiben

$$f_{a,b}(x) = ax + b$$

Hier ist x ein Argument, a und b sind Parameter. Diese Schreibweise hat verschiedene Nachteile:

- Jede Funktion braucht einen Namen.
- Manchmal wird eine Funktion auf unterschiedliche Weisen betrachtet (Argument/Parameter) $g_b(a, x) = f_{a,b}(x)$.

Wenn in einem Anwendungsgebiet über wenige Funktionen Aussagen gemacht werden, kann das in Kauf genommen werden, wenn aber eine Theorie aufgebaut werden soll, die über Funktionen spricht, ist die obige Formulierung ungeeignet.

2.1.1 Die λ-Schreibweise

Church [Chu32] hat folgende Schreibweise eingeführt.

$$f = \lambda x.ax + b$$

Die Funktionsapplikation, d.h. die Anwendung einer Funktion auf ihr Argument, wird als Juxtapunktion und Funktionsabstraktion mit λ, Standardoperationen wie $+$ und $\cdot$ oft wie oben infix geschrieben. Genauer wäre

$$f = \lambda x.(+(\cdot ax)b)$$

zu schreiben. Statt $f_{a,b}(5)$ schreibt wir jetzt also $(f5)$. Wenn wir die Funktion f nun auch als Funktion von a sehen wollen, schreiben wir $\lambda a.\lambda x.ax + b$ oder einfacher $\lambda a, x.ax + b$.

Formal sehen die möglichen Ausdrücke des Lambdakalküls so aus:

$$E = x \mid c \mid EF \mid \lambda x.E$$

x steht dabei für eine beliebige Variable, c für eine Konstante EF für die Anwendung der Funktion E auf das Argument F und $\lambda x.E$ für eine Funktion, die angewandt auf x das Ergebnis E hat. Die Priorität der Operationen ist so geregelt, daß die Applikation linksassoziativ ist und, daß sich E in $\lambda x.E$ soweit wie möglich nach rechts erstreckt. Es gilt also

$$\lambda x.EF = \lambda x.(EF), \quad EFG = (EF)G.$$

2.1.2 Reduktion

In der alten Schreibweise war

$$f_{a,b}(5) = 5a + b$$

Im Lambdakalkül führen wir die β-Reduktion ein um Berechnungen zu beschreiben:

$$(\lambda x.E)F \rightarrow_\beta [F/x]E$$

Wir schreiben $[F/x]E$ für den Term, der entsteht, wenn wir jedes freie Vorkommen von x in E durch F ersetzen. Dabei ist zu beachten, daß die gebundenen Variablen in E verschieden von den freien Variablen in F zu wählen sind. Es entstehen dann keine zusätzlichen Bindungen.

Um dieses ganz zu verstehen, müssen wir zuerst die freien Variablen $\mathrm{fv}(E)$ eines Ausdrucks E definieren:

$$\mathrm{fv}(x) = \{x\}; \mathrm{fv}(c) = \emptyset; \mathrm{fv}(EF) = \mathrm{fv}(E) \cup \mathrm{fv}(F);$$
$$\mathrm{fv}(\lambda x.E) = \mathrm{fv}(E) \setminus \{x\}$$

Alle anderen Variablen heißen gebunden.

Außerdem sollen $\lambda x.x$ und $\lambda y.y$ offensichtlich die gleiche Bedeutung haben. Zwei Ausdrücke, die sich nur in den Namen gebundener Variablen unterscheiden, heißen α-äquivalent. Wir wollen, wenn wir einen Ausdruck E angeben, immer seine α-Äquivalenzklasse meinen.

Wir können nun zum Beispiel

$$(\lambda x.ax + b)5 \rightarrow_\beta a \cdot 5 + b$$

reduzieren. Wir nennen eine Relation $\rightarrow$ kompatibel, falls mit $F \rightarrow F'$ immer auch $E[F] \rightarrow E[F']$ gilt. $E[F]$ bedeutet dabei, daß F ein Teilausdruck von $E[F]$ an einer bestimmten Stelle ist. Bei $E[F']$ steht dann an dieser Stelle F'. Wir können das nicht mit Substitutionen formulieren, da in F Variablen frei vorkommen dürfen, die in $E[F]$ gebunden sind.

Wir schreiben $\rightarrow_\beta^*$ für die reflexive, transitive und kompatible Hülle von $\rightarrow_\beta$ und $=_\beta$ für die reflexive, transitive, kompatible und symmetrische Hülle.

2.1.3 Church-Kodierung

Wir können im Lambdakalkül natürliche Zahlen und Wahrheitswerte modellieren. Das heißt, es gibt Lambdaausdrücke $\Lambda(0)$, $\Lambda(1)$,$\Lambda(+)$,$\Lambda(\mathrm{true})$,$\Lambda(\mathrm{false})$, so daß die üblichen Rechenregeln gelten, also zum Beispiel

$$\Lambda(+)\Lambda(m)\Lambda(n) =_\beta \Lambda(m+n)$$

Eine spezielle Kodierung mit dieser Eigenschaft heißt Church-Kodierung [Chu32]. Zum Kodieren von mehrstelligen Funktionen verwenden wir *currying*, d.h. wir schreiben + als Funktion mit einem Argument, deren Resultat wieder eine Funktion ist. (+3) ist dann eine Funktion, die 3 zu ihrem

Argument addiert. Diese Methode stammt von Schönfinkel [Sch24]. Es ist

$$\begin{aligned}
\Lambda(0) &= \lambda x.\lambda f.x \\
\Lambda(1) &= \lambda x.\lambda f.fx \\
\Lambda(\texttt{succ}) &= \lambda n.\lambda x.\lambda f.f(nxf) \\
\Lambda(+) &= \lambda n.\lambda m.n(\lambda m.m)(\lambda f.\lambda m.\Lambda(\texttt{succ})(fm))m \\
\Lambda(\texttt{true}) &= \lambda x.\lambda y.x \\
\Lambda(\texttt{false}) &= \lambda x.\lambda y.y \\
\Lambda(\texttt{iszero}) &= \lambda n.n\Lambda(\texttt{true})(\lambda z.\Lambda(\texttt{false}))
\end{aligned}$$

`true` und `false` sind so gewählt, daß `true x y` $=_\beta$ `x` und `false x y` $=_\beta$ `y` gilt.

Natürliche Zahlen und Wahrheitswerte auf diese Weise zu repräsentieren, wäre aber mühsam. Für praktische Zwecke führen wir 0, 1, + usw. als Konstanten ein. Da aber (+ 3 4) und 7 in gewissem Sinne gleich sein sollen, führen wir noch eine weitere Reduktionsregel, die δ-Regel ein. Diese führt dann Funktionsanwendungen mit festgelegten Funktionen und konstanten Argumenten durch. Also

$$(+\ 3\ 4) \rightarrow_\delta 7$$

2.1.4 Fixpunkt und Rekursion

Wir können den Fixpunktkombinator Y mit folgender Eigenschaft definieren.

$$Yf =_\beta f(Yf)$$

Eine mögliche Definition von Y sieht so aus:

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

Mit Y können wir eine Funktion f, für die die Gleichung

$$f =_\beta M$$

gelten soll (f kommt frei in M vor), auch durch die nicht rekursive Form

$$f = Y(\lambda f.M)$$

beschreiben (f tritt rechts nur gebunden auf). Als Beispiel, wie mit Hilfe von Y rekursive Funktionen definiert werden können, sehen wir uns die folgende Definition der Fakultät an:

$$\texttt{fak} = Y(\lambda f.\lambda n.(\texttt{iszero}\ \ n)1(n * f(n-1)))$$

In einigen der folgenden typisierten Kalküle ist der Typ von Y zwar beschreibbar, die Definition von Y aber nicht typisierbar. Y muß dann explizit als Konstrukt der Programmiersprache eingeführt werden.

2.1.5 Semantik

Mit Hilfe der Reduktion können wir nun bereits eine operationelle Semantik für Ausdrücke angeben, die Äquivalenzklasse bezüglich $=_\beta$. Dennoch will ist daneben oft eine denotationelle Semantik erwünscht. Der Grund ist, daß wir Äquivalenz oft weiter fassen wollen. Ein Optimierer soll die Funktion f schon durch g ersetzen dürfen, wenn $fx =_\beta gx$ für alle x gilt, selbst wenn $f \neq_\beta g$ ist. Ein Beispiel ist die η-Reduktion

$$\lambda x.(Ex) \to_\eta E, \quad x \notin \mathrm{fv}(E).$$

Diese erlauben wir, wenn in der denotationellen Semantik $[\![\lambda x.(Ex)]\!] = [\![E]\!]$ gilt.

Es gibt solche denotationellen Semantiken für den ungetypten Lambdakalkül, die jedem λ-Ausdruck eine Semantik zuordnen [Sco76], [Sco82], aber in Programmiersprachen werden in der Regel andere Semantiken betrachtet [Mil78], [Gun92]. Diese Semantiken sind nicht für alle Terme definiert, bzw. liefern manchmal den speziellen Wert wrong. Für $1 + (\lambda x.x)$ zum Beispiel liefert Milners Semantik [Mil78] den Wert wrong, da $[\![\lambda x.x]\!]$ keine ganze Zahl ist. Wir wollen nun sicherstellen, daß zur Laufzeit solche Terme nicht entstehen. Dazu werden wir im folgenden Typsysteme einführen, die genau das vermeiden.

2.2 Der einfach getypte Lambdakalkül

Das Problem des ungetypten Lambdakalküls ist, daß zum Beispiel

$$1 + (\lambda x.x)$$

ein korrekter Term ist. Typen werden eingeführt, um solche Ausdrücke zu verbieten. Im Fall des einfach getypten Lambdakalküls sehen die Typen folgendermaßen aus:

$$\tau = \iota \mid \alpha \mid \tau \to \tau$$

ι sei dabei ein Basistyp wie `Int`, α eine Typvariable und $\tau_1 \to \tau_2$ sei ein Funktionstyp mit dem Argumenttyp τ_1 und dem Resultattyp τ_2.

Für die Frage, ob ein Term typkorrekt ist, kommt es auch auf den Kontext an, also darauf, von welchem Typ die freien Variablen sind. Wir beschreiben den Kontext formal durch eine Folge Γ von Variablendeklarationen. Ein Ausdruck E hat dann den Typ τ im Kontext Γ, wenn $\Gamma \vdash E : \tau$ ein gültiges Urteil ist. Gültige Urteile werden über folgendes Regelsystem definiert:

$$(var)\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau}$$

$$(const_c)\frac{}{\Gamma \vdash c : \tau_c} \quad \text{für jede Konstante } c \text{ des Kalküls}$$

$$(\to I)\frac{\Gamma, x : \tau \vdash E : \tau'}{\Gamma \vdash \lambda x : \tau.E : \tau \to \tau'}$$

$$(\to E)\frac{\Gamma \vdash E : \tau \to \tau' \quad \Gamma \vdash F : \tau}{\Gamma \vdash EF : \tau'}$$

Wir gehen in $(\to I)$ davon aus, daß x in Γ noch nicht deklariert ist. Da wir Ausdrücke nur modulo α-Äquivalenz betrachten ist das keine Einschränkung.

Statt der hier verwendeten Formulierung nach Church [Chu40] kann auch die von Curry [Cur34] bevorzugte verwendet werden.

$$(\to I')\frac{\Gamma, x : \tau \vdash E : \tau'}{\Gamma \vdash \lambda x.E : \tau \to \tau'}$$

Aus einer Typisierung nach Church kann offensichtlich durch Streichung des Typs im λ-Term eine Typisierung nach Curry gewonnen werden, aber auch umgekehrt kann man zu jeder Curry-Typisierung eine Church-Typisierung finden, aus der durch Streichung die ursprüngliche Curry-Typisierung entsteht. Die rekonstruierten Typen sind jedoch nicht immer eindeutig. In

$$(\lambda y.\lambda z.z)(\lambda x.x)3 : \texttt{Int}$$

kann für x ein beliebiger Typ gewählt werden. Wir wollen uns hier an die Formulierung nach Church halten und Auslassungen von Typen im Abschnitt über Typrekonstruktion diskutieren. Die Regel $(const_c)$ wird später von algebraischen Datentypen subsumiert und auch sonst der Einfachheit manchmal weggelassen.

Ein λ-Ausdruck ist in Normalform, wenn er nicht weiter reduziert werden kann. Der einfach getypte Lambdakalkül ist stark normalisierend, d.h. $\rightarrow_\beta$ ist noethersch (es gibt keine unendlich langen Ableitungen), und in jeder β-Äquivalenzklasse gibt es genau eine Normalform. Diese Eigenschaft geht verloren, wenn wir eine Familie von Konstanten $\mathbf{fix}_\tau$ einführen

$$(fix)\frac{}{\Gamma \vdash \mathbf{fix}_\tau : (\tau \to \tau) \to \tau}$$

und die Reduktion

$$\mathbf{fix}_\tau f \to f(\mathbf{fix}_\tau f)$$

zulassen.

2.3 Algebraische Typen

Wenn wir den Lambdakalkül in der Praxis verwenden, wollen wir eigene Datentypen definieren können. Wir haben gesehen, daß wir bei den natürlichen Zahlen eine Church-Kodierung verwenden können, haben uns aber entschieden, diese lieber als Konstanten in den Kalkül aufzunehmen. Wenn wir aber auch benutzerdefinierte Typen und Konstanten zulassen wollen, müssen wir entweder wieder auf eine Church-Kodierung zurückgreifen oder aber andere Möglichkeiten finden, neue Typen zu bauen.

Die Menge der natürlichen Zahlen kann als die kleinste Menge mit der Eigenschaft

$$\mathtt{nat} = \{0\} \oplus \{\mathtt{succ}(x) \mid x \in \mathtt{nat}\}$$

definiert werden. Hier haben wir zur Definition disjunkte Vereinigung (im folgenden auch Summe genannt) und Rekursion verwendet. Wir wollen nun diese zwei fundamentalen Konstruktionen und zusätzlich Produkte auf Typen erlauben. Die Möglichkeit für den Benutzer, solche eigene, algebraische Typen zu notieren, ist die folgende:

$$\mathbf{data}\ \mathtt{T} = D_1\tau_{1,1}\ldots\tau_{1,k_1} \mid \ldots \mid D_j\tau_{j,1}\ldots\tau_{j,k_j};$$

Hier wird der Typ `T` (eventuell rekursiv) definiert. | entspricht der Summe, wobei die D_i markieren, aus welchem Summand ein Element stammt. Jeder Summand entspricht dem Produkt der $\tau_{i,j}$ (i fest).

Die natürlichen Zahlen würden wir also beispielsweise als

data `Nat` = `Zero` | `Succ Nat`;

definieren.

Bei Funktionstypen haben wir die Abstraktion, um Ausdrücke mit Funktionstyp zu erzeugen, und die Applikation, um den Funktionstyp zu eliminieren. Ebenso geben wir hier Regeln an, um algebraische Typen einzuführen und zu eliminieren:

$$(\textbf{data } I)\frac{\textbf{data } \texttt{T} = \ldots \mid \texttt{D } \tau_1 \ldots \tau_k \mid \ldots}{\texttt{D} : \tau_1 \to \ldots \to \tau_k \to \texttt{T}}$$

Für die Destruktion wird die case-Konstruktion verwendet:

$$(\textbf{data } E)\frac{\begin{array}{l}\textbf{data } \texttt{T} = \sum_i D_i \bar{\tau}_i \\ (\Gamma, \bar{y}_i : \bar{\tau}_i) \vdash E_i : \tau \\ \Gamma \vdash F : \texttt{T}\end{array}}{\Gamma \vdash \textbf{case } F \textbf{ of } \{D_i \bar{y}_i \Rightarrow E_i\} : \tau}$$

Wenn `Nat` wie oben definiert ist, kann die Fakultät rekursiv so definiert werden (wir schreiben `\x -> E` für $\lambda x.E$):

```
fak = \m ->  case m of
   Zero -> 1;
   Succ n -> (* n (fak (- n 1)));
```

Wir können es aber auch mit Hilfe des Y-Kombinators, der in Programmiersprachen oft **fix** geschrieben wird, notieren:

```
fak = fix \f -> \m -> case m of
   Zero -> 1;
   Succ n -> (* n (f (- n 1)));
```

Ein Beispiel, in dem ein echtes Produkt vorkommt, wäre eine Liste ganzer Zahlen:

```
data IntList = Nil | Cons Int IntList;
```

Wir nehmen dabei `Int` als gegeben an.

2.4 Polymorphie

Wenn wir nun zum Beispiel die Länge einer `IntList` berechnen wollen, geht das rekursiv so:

```
lenint = \l -> case l of
   Nil -> 0;
   Cons i l2 -> (+ 1 (lenint l2));
```

Nehmen wir aber an, wir haben auch noch Listen von Wahrheitswerten und wollen auch deren Länge berechnen, so müssen wir dafür eine weitere Typdeklaration und auch eine weitere Längenfunktion angeben. Um das zu vermeiden, parameterisieren wir den Listentyp wie folgt:

```
data List a = Nil | Cons a (List a);
```

Die Längendefinition

```
len = \l -> case l of
   Nil -> 0;
   Cons x l2 -> (+ 1 (len l2));
```

funktioniert jetzt für alle Listen. Wenn wir nun aber den Typ von `len` ansehen, stellen wir fest, daß `len` sowohl den Typ `(List Int) -> Int` als auch den Typ `(List Bool) -> Int` hat. Eine solche Funktion, die mehrere Typen hat, heißt polymorph.

Bevor wir uns formal mit Polymorphie auseinandersetzen, wollen wir die verschiedenen Arten der Polymorphie charakterisieren. Die Ideen dieser Charakterisierung finden sich in der Literatur ([CW85], [Bar92], [Mit90] und [Str67]).

2.4.1 Scheinbare und echte Polymorphie

In manchen Programmiersprachen darf die Multiplikation `*` sowohl für ganze als auch für rationale Zahlen verwendet werden. Dies kann vom Übersetzer erkannt, und sofort durch die passende Multiplikation ersetzt werden. Der Programmierer spart hier, sich die Namen für verschiedene Multiplikationen zu merken und muß nicht auf die gewohnte mathematische Notation verzichten. Dies wird als *scheinbare* Polymorphie bezeichnet.

Von *echter* Polymorphie [CW85] wollen wir erst dann sprechen, wenn der Benutzer neue polymorphe Funktionen, wie zum Beispiel eine Exponentiation, mit Hilfe der polymorphen Multiplikation schreiben kann. Welche Multiplikation in der Exponentiation verwendet werden soll, kann nur dynamisch entschieden werden. Der Programmierer muß jetzt die Exponentiationsfunktion nicht mehrmals hinschreiben.

2.4.2 Uniforme und ad-hoc Polymorphie

Die Längenfunktion auf Listen ist ein typisches Beispiel für *uniforme* Polymorphie. Die Operation arbeitet uniform auf Listen und greift auf Komponenten des Parametertyps nicht zu.

Die Multiplikation hingegen führt zwar auf ganzen Zahlen und Fließkommazahlen eine verwandte Operation aus, ruft aber, je nach Argumenttyp, unterschiedliche Funktionen auf. Dies wird als *ad-hoc*-Polymorphie bezeichnet [Str67].

2.4.3 Subtyp- und parametrische Polymorphie

Bedingt durch die Polymorphie kann ein Ausdruck mehrere Typen haben. Dies kann einerseits dadurch entstehen, daß wir Aussagen der Form „jede ganze Zahl ist auch eine rationale Zahl“ im Typsystem reflektieren, also ein Ausdruck vom Typ `Int` automatisch auch vom Typ `Rat` ist. Dies wird dann *Subtyppolymorphie* genannt.

Den Gegenpol bildet die *parametrische* Polymorphie, bei der Parameter in Typschemata verschieden ersetzt werden können, und dadurch verschiedene Typen entstehen [CW85].

Manchmal wird Subtyppolymorphie auch *vertikale* Polymorphie und parametrische Polymorphie *horizontale* Polymorphie genannt.

2.4.4 Explizite und implizite Polymorphie

In manchen polymorphen Typsystemen gibt es keinen Term, der echt zwei Typen hat. Zum Beispiel spiegelt sich jede Spezialisierung von Typparametern im Term wieder. Wir schreiben

```
(len Int) [1,2,3]
```

statt

```
len [1,2,3]
```

Subtyppolymorphie wird durch eine Einbettungsfunktion im Term ausgedrückt. Das wird *expliziter* Polymorphie genannt. Falls manche dieser Angaben fehlen, wird von *impliziter* Polymorphie gesprochen [Bar92].

2.4.5 Prädikative und imprädikative Polymorphie

Wenn wir parametrisierte Typen zulassen, können wir zwischen monomorphen Typen (ohne Parameter) und polymorphen Typen oder Typschemata (mit Parametern) unterscheiden. Dürfen wir für Parameter Typschemata einsetzen oder nur monomorphe Typen? Imprädikative, wie System F [Rey74], [Gir71] oder der „Calculus of Constructions“ [CH88] lassen Typschemata zu. Prädikativen Systemen lassen entweder, wie ML [MTH90], nur monomorphe Typen zu oder behelfen sich, wie das System von Martin-Löf [ML84], [NPS90], mit einer Hierarchie, so daß in Typschemata der Stufe n nur Typschemata der Stufe $n-1$ eingesetzt werden können. Diese Unterscheidung wird von Mitchell [Mit90] genauer ausgeführt.

2.5 Der λ-Würfel

Nachdem wir verschiedene Arten der Polymorphie charakterisiert haben, wollen wir uns jetzt formal mit parametrischer Polymorphie auseinandersetzen. Die Diskussion orientiert sich an Barendregt [BH90], [Bar92].

In unserer Diskussion haben wir zwei getrennte Universen: Typen und Terme. Wir haben Terme, in denen Termvariablen vorkommen, und wir können mit Hilfe von λ-Ausdrücken über diese Termvariablen abstrahieren. Bisher können wir allerdings in einem Term keine Typvariablen verwenden. Nur konstante Typen dürfen vorkommen, zum Beispiel in der Identitätsfunktion auf den ganzen Zahlen:

$$\lambda x : \texttt{Int}.x$$

Es liegt nahe, auch Typvariablen zuzulassen, also zum Beispiel $(\lambda x : t.x) : t \to t$ und auch darüber zu abstrahieren:

$$\Lambda t.(\lambda x : t.x) : \forall t.(t \to t)$$

Wir müssen hier t sowohl im Term als auch im Typ binden, oft wird dazu in der Literatur auch Π statt des Allquantors verwendet. Wir verwenden $\forall$ und später auch $\exists$, da für uns der Bezug zu den logischen Quantoren später eine Rolle spielt.

Ähnlich können wir auch Typ-Typ-Abhängigkeiten beziehungsweise Typ-Term-Abhängigkeiten modellieren. Diese drei Erweiterungen sind unabhängig und jede beliebige Kombination macht Sinn. Es entsteht ein Würfel mit den acht möglichen Typsystemen an den Ecken, wobei jede Dimension für eine Erweiterung steht. Dies ist der sogenannte λ-Würfel.

2.5.1 Terme, die von Typen abhängen

Die Identitätsfunktion ist ein gutes Beispiel für eine Funktion, die von einem Typ abhängt. Wir möchten über Typen abstrahieren und wie oben haben:

$$\vdash \Lambda t.(\lambda x : t.x) : \forall t.(t \to t)$$

Für korrekt getypte Terme geben wir Regeln an. Wir brauchen das, da nicht jeder syntaktisch korrekte Term auch typkorrekt ist. Bei Typen reicht im Moment noch eine syntaktische Charakterisierung aus:

$$\tau = \alpha \mid \iota \mid \tau \to \tau \mid \forall \alpha.\tau$$

Gegenüber dem einfach getypten Lambdakalkül kommen hier Typschemata hinzu. Unser Church Regelsystem müssen wir daher um zwei Regeln erweitern:

$$(var)\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau}$$

$$(\to I)\frac{\Gamma, x : \tau \vdash E : \tau'}{\Gamma \vdash \lambda x : \tau.E : \tau \to \tau'}$$

$$(\to E)\frac{\Gamma \vdash E : \tau \to \tau' \quad \Gamma \vdash F : \tau}{\Gamma \vdash EF : \tau'}$$

$$(\forall E)\frac{\Gamma \vdash E : \forall t.\tau'}{\Gamma \vdash (E\tau) : [\tau/t]\tau'}$$

$$(\forall I)\frac{\Gamma \vdash E : \tau}{\Gamma \vdash \Lambda t.E : \forall t.\tau}$$

Außerdem müssen wir die β-Reduktion um die Reduktion mit Typparametern erweitern:

$$(\Lambda t.E)\tau \rightarrow_\beta [\tau/t]E$$

Dieser Kalkül erlaubt jetzt beispielsweise eine polymorphe Definition der Funktionskomposition.

$$\begin{array}{l}(\Lambda t_1.\Lambda t_2.\Lambda t_3.\lambda f : (t_1 \rightarrow t_2).\lambda g : (t_2 \rightarrow t_3).\lambda x : t_1.g(fx)) : \\ \quad \forall t_1.\forall t_2.\forall t_3.(t_1 \rightarrow t_2) \rightarrow (t_2 \rightarrow t_3) \rightarrow t_1 \rightarrow t_3\end{array}$$

Dieses Typsystem wird oft auch als System F oder F_2 bezeichnet und wurde zuerst von Girard [Gir71] und Reynolds [Rey74] untersucht.

2.5.2 Typen, die von Typen abhängen

Der Typ $t \rightarrow t$ ist für alle t ein Typ. Es ist nun naheliegend, auch Typen über Typen zu abstrahieren und das Konstrukt $\Lambda t : \mathbf{type}.t \rightarrow t$ einzuführen. Ein anderes Beispiel für einen Typkonstruktor dieser Art ist $\Lambda t : \mathbf{type}.(\texttt{List}\ t)$. Der Typkonstruktor hat die *Art* (engl. Kind) $\mathbf{type} \rightarrow \mathbf{type}$ und wir schreiben

$$(\Lambda t : \mathbf{type}.(\texttt{List}\ t)) : \mathbf{type} \rightarrow \mathbf{type}.$$

Wir müssen nun auch Bildungsregeln für Typen angeben, da $\Lambda t : \mathbf{type}.t \rightarrow t$ als Typ für einen Wert der Programmiersprache nicht in Frage kommt. Wir müssen die Typen unter den Typkonstruktoren auszeichnen, es sind die der Art **type**. Für Arten reicht wieder eine syntaktische Definition:

$$\kappa = \mathbf{type} \mid \kappa \rightarrow \kappa$$

Im Kontext müssen wir nun nicht nur Termvariablen- sondern auch Typkonstruktorvariablen deklarieren. Wichtig ist nun auch, daß ein Typ deklariert wird, bevor er in einer Variablendeklaration benutzt wird, also müssen wir auf die Reihenfolge im Kontext achten. Für unseren bisherigen Kalkül hätten wir statt der Variablenregel die zwei folgenden Regeln einführen können:

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

$$(weak)\frac{\Gamma \vdash x : \tau}{\Gamma, y : \tau' \vdash x : \tau}$$

Wir brauchen nun Regeln für die Einführung beider Arten von Variablen und alle vier Möglichkeiten zur Abschwächung:

$$(var_1)\frac{\Gamma \vdash \tau : \mathbf{type}}{\Gamma, x : \tau \vdash x : \tau} \quad x \text{ new}$$

$$(var_2)\frac{}{\Gamma, t : \kappa \vdash t : \kappa} \quad t \text{ new}$$

$$(weak_1)\frac{\Gamma \vdash E : \tau' \quad \Gamma \vdash \tau : \mathbf{type}}{\Gamma, x : \tau \vdash E : \tau'} \quad x \text{ new}$$

$$(weak_2)\frac{\Gamma \vdash \tau' : \kappa \quad \Gamma \vdash \tau : \mathbf{type}}{\Gamma, x : \tau \vdash \tau' : \kappa} \quad x \text{ new}$$

$$(weak_3)\frac{\Gamma \vdash E : \tau'}{\Gamma, t : \kappa \vdash E : \tau'} \quad t \text{ new}$$

$$(weak_4)\frac{\Gamma \vdash \tau' : \kappa}{\Gamma, t : \kappa' \vdash \tau' : \kappa} \quad t \text{ new}$$

Es gibt Abstraktion und Applikation auf Term- und auf Typebene:

$$(\to F)\frac{\Gamma \vdash \tau : \mathbf{type} \quad \Gamma \vdash \tau' : \mathbf{type}}{\Gamma \vdash \tau \to \tau' : \mathbf{type}}$$

$$(\to E)\frac{\Gamma \vdash E : \tau \to \tau' \quad \Gamma \vdash F : \tau}{\Gamma \vdash EF : \tau'}$$

$$(\to I)\frac{\Gamma, x : \tau \vdash E : \tau'}{\Gamma \vdash \lambda x : \tau.E : \tau \to \tau'}$$

$$(\to E_2)\frac{\Gamma \vdash \tau : \kappa \to \kappa' \quad \Gamma \vdash \tau' : \kappa}{\Gamma \vdash \tau\tau' : \kappa'}$$

$$(\to I_2)\frac{\Gamma, t : \kappa \vdash \tau : \kappa'}{\Gamma \vdash \Lambda t : \kappa.\tau : \kappa \to \kappa'}$$

Zuletzt brauchen wir die β-Regel auf Typebene, damit $(\Lambda t : \mathbf{type}.t)\texttt{Int}$ und `Int` derselbe Typ sind.

$$(conv)\frac{\Gamma \vdash E : \tau \quad \Gamma \vdash \tau' : \mathbf{type}}{\Gamma \vdash E : \tau'} \quad \tau =_\beta \tau'$$

Wir führen nun noch zusätzliche Regeln für die Bildung von Arten ein, um anschließend das Ganze zu vereinheitlichen. $\Gamma \vdash \kappa : \mathbf{kind}$ steht für eine korrekt gebildete Art. $(axiom)$ und $(\to F_2)$ entsprechen den syntaktischen Regeln. Ansonsten kommen im wesentlichen neue Abschwächungsregeln hinzu.

$$(axiom) \vdash \mathbf{type} : \mathbf{kind}$$

$$(var'_2)\frac{\Gamma \vdash \kappa : \mathbf{kind}}{\Gamma, t : \kappa \vdash t : \kappa} \quad t \text{ new}$$

$$(weak'_3)\frac{\Gamma \vdash E : \tau \quad \Gamma \vdash \kappa : \mathbf{kind}}{\Gamma, t : \kappa \vdash E : \tau}$$

$$(weak'_4)\frac{\Gamma \vdash \tau : \kappa' \quad \Gamma \vdash \kappa : \mathbf{kind}}{\Gamma, t : \kappa \vdash \tau : \kappa'}$$

$$(weak_5)\frac{\Gamma \vdash \kappa : \mathbf{kind} \quad \Gamma \vdash \tau : \mathbf{type}}{\Gamma, x : \tau \vdash \kappa : \mathbf{kind}}$$

$$(weak_6)\frac{\Gamma \vdash \kappa' : \mathbf{kind} \quad \Gamma \vdash \kappa : \mathbf{kind}}{\Gamma, t : \kappa \vdash \kappa' : \mathbf{kind}}$$

$$(\to F_2)\frac{\Gamma \vdash \kappa : \mathbf{kind} \quad \Gamma \vdash \kappa' : \mathbf{kind}}{\Gamma \vdash \kappa \to \kappa' : \mathbf{kind}}$$

Das nun entstandene Regelsystem enthält viel Redundanz. Wir führen deshalb für **type** und **kind** den Begriff Sorte ein und formulieren dieselben Regeln, wobei s über Sorten läuft.

$$(axiom) \vdash \mathbf{type} : \mathbf{kind}$$

$$(var)\frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A} \quad x \text{ new}$$

$$(weak)\frac{\Gamma \vdash B : C \quad \Gamma \vdash A : s}{\Gamma, x : A \vdash B : C}$$

$$(\to F)\frac{\Gamma \vdash A : s \quad \Gamma \vdash B : s}{\Gamma \vdash A \to B : s}$$

$$(\to E)\frac{\Gamma \vdash E : A \to B \quad \Gamma \vdash F : A}{\Gamma \vdash EF : B}$$

$$(\to I)\frac{\Gamma \vdash A : s \quad \Gamma \vdash B : s \quad \Gamma, x : A \vdash E : B}{\Gamma \vdash \lambda x : A.E : A \to B}$$

$$(conv)\frac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s}{\Gamma \vdash A : B'} \quad B =_\beta B'$$

Diese Formulierung ist knapper und übersichtlicher, legt uns aber darauf fest, die verschiedenen Sorten möglichst analog zu behandeln. Hier dürfen Terme nur von Termen und Typen nur von Typen abhängen. Dies wird in $(\to F)$ dadurch festgelegt, daß A und B von der selben Sorte sein müssen.

Wir haben hier auf die im vorigen Abschnitt behandelte Möglichkeit, Terme die von Typen abhängen zu beschreiben verzichtet. Das hier gegebene Typsystem wird mit $\lambda\underline{\omega}$ bezeichnet. Die Kombination von $\lambda\underline{\omega}$ mit F heißt dann F_ω [Gir71].

2.5.3 Typen, die von Termen abhängen

Dies ist für uns besonders interessant, da es eine Alternative zu indizierten Typen darstellt. Beispiele sind hier Vektoren oder Matrizen und ihre Länge. In der Literatur werden diese Typen oft einfach *abhängige* Typen genannt. Die ($\forall$)-Regeln sind als Verallgemeinerung der ($\rightarrow$)-Regeln zu lesen. $A \rightarrow B$ ist ein Spezialfall von $\forall x : A.B$, wenn $x \notin \mathrm{fv}(B)$.

$$(axiom) \vdash \mathbf{type} : \mathbf{kind}$$

$$(var)\frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A} \quad x \text{ new}$$

$$(weak)\frac{\Gamma \vdash B : C \quad \Gamma \vdash A : s}{\Gamma, x : A \vdash B : C}$$

$$(\forall F)\frac{\Gamma \vdash A : \mathbf{type} \quad \Gamma, x : A \vdash B : s}{\Gamma \vdash (\forall x : A.B) : s}$$

$$(\forall E)\frac{\Gamma \vdash E : (\forall x : A.B) \quad \Gamma \vdash F : A}{\Gamma \vdash EF : [F/x]B}$$

$$(\forall I)\frac{\Gamma \vdash A : \mathbf{type} \quad \Gamma, x : A \vdash B : s \quad \Gamma, x : A \vdash E : B}{\Gamma \vdash \lambda x : A.E : (\forall x : A.B)}$$

$$(conv)\frac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s}{\Gamma \vdash A : B'} \quad B =_\beta B'$$

Die ($\forall F$) Regel erlaubt hier nur Abhängigkeiten von Termen, was durch $A : \mathbf{type}$ beschrieben wird.

Abhängige Typen sind für Beweiser besonders interessant, da sie, wie wir später sehen werden, beim Curry-Howard-Isomorphismus Eigenschaften entsprechen, die mit Werten parametrisiert sind.

2.5.4 Alle Abhängigkeiten

Wir beschreiben hier nun ein parametrisiertes System, das je nach Parametrisierung alle in diesem Abschnitt vorgestellten Systeme enthält. Wir werden sogar noch allgemeiner, da wir statt $\{\mathbf{type}, \mathbf{kind}\}$ eine allgemeine Menge S von Sorten annehmen. In vielen Fällen ist eine Sorte **prop** interessant, um Aussagen und andere Typen unterschiedlich zu behandeln. Welche Axiome wir haben und welche Regeln, wird durch A und R bestimmt. In dieser Allgemeinheit wird von *pure type systems* gesprochen.

$$(axiom)\frac{}{\vdash c : s} \quad (c : s) \in A$$

$$(var)\frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A} \quad s \in S, x \,\text{new}$$

$$(weak)\frac{\Gamma \vdash B : C \quad \Gamma \vdash A : s}{\Gamma, x : A \vdash B : C}$$

$$(\forall F)\frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (\forall x : A.B) : s_3} \quad (s_1, s_2, s_3) \in R$$

$$(\forall E)\frac{\Gamma \vdash E : (\forall x : A.B) \quad \Gamma \vdash F : A}{\Gamma \vdash EF : [F/x]B}$$

$$(\forall I)\frac{\Gamma \vdash (\forall x : A.B) : s \quad \Gamma, x : A \vdash E : B}{\Gamma \vdash \lambda x : A.E : (\forall x : A.B)}$$

$$(conv)\frac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s}{\Gamma \vdash A : B'} \quad B =_\beta B', s \in S$$

An der Formationsregel $(\forall F)$ ist hier zu sehen, wie durch die Relation R die verschiedenen bisher behandelten Möglichkeiten ins Spiel kommen. Die oben besprochenen Systeme hatten $S = \{\mathbf{type}, \mathbf{kind}\}$, $A = \{\mathbf{type} : \mathbf{kind}\}$ und $R = \{(\mathbf{type}, \mathbf{type}, \mathbf{type})\}$ für den einfach getypten Lambdakalkül, $R = \{(\mathbf{type}, \mathbf{type}, \mathbf{type}), (\mathbf{kind}, \mathbf{type}, \mathbf{type})\}$ für Terme, die von Typen

abhingen, $R = \{(\mathbf{type}, \mathbf{type}, \mathbf{type}), (\mathbf{kind}, \mathbf{kind}, \mathbf{kind})\}$ für Typen, die von Typen abhingen und zuletzt $R = \{(\mathbf{type}, \mathbf{type}, \mathbf{type}), (\mathbf{type}, \mathbf{kind}, \mathbf{kind})\}$ für Typen, die von Termen abhingen. Die Relation

$$R = \{(\mathbf{type}, \mathbf{type}, \mathbf{type}), (\mathbf{kind}, \mathbf{type}, \mathbf{type}), \\ (\mathbf{kind}, \mathbf{kind}, \mathbf{kind})\}$$

gibt gerade F_ω, mit

$$R = \{(\mathbf{type}, \mathbf{type}, \mathbf{type}), (\mathbf{kind}, \mathbf{type}, \mathbf{type}), \\ (\mathbf{kind}, \mathbf{kind}, \mathbf{kind}), (\mathbf{type}, \mathbf{kind}, \mathbf{kind})\}$$

bekommen wir gerade den *Calculus of Constructions* [CH88].

2.6 Erweiterungen des Würfels

Insbesondere bei mehreren Sorten und zusätzlichen Termkonstruktionen ist ein andere Betrachtungsweise interessant. Jacobs [Jac91], [Jac96] schlägt vor, eine semantisch motivierte Abhängigkeit zwischen Sorten einzuführen. Um eine bestimmte Regel oder ein *feature*, welches mit Sorten parametrisiert ist, zuzulassen, müssen bestimmte Abhängigkeiten zwischen den involvierten Sorten gelten. Im System von Jacobs fällt der Würfel auf zwei Systeme zusammen, es können aber Systeme beschrieben werden, die von *pure type systems* nicht erfaßt werden.

Die wichtigste Regel, die wir noch hinzufügen wollen, ist die Summe:

$$(\exists F)\frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \exists x : A.B : s_2}$$

$$(\exists I)\frac{\Gamma \vdash E : A \quad \Gamma \vdash F : [E/x]B \quad \Gamma \vdash \exists x : A.B : s}{\Gamma \vdash \langle E, F\rangle : \exists x : A.B}$$

Für die Elimination von Summen gibt es zwei Möglichkeiten, eine für sogenannte schwache Summen,

$$(\exists_{weak} E)\frac{\Gamma \vdash E : \exists x : A.B \quad \Gamma \vdash C : s \quad \Gamma, x : A, y : B \vdash F : C}{\Gamma \vdash (F \ \mathbf{where}\ \langle x, y\rangle = E) : C}$$

und eine für starke Summen:

$$(\exists_{strong}E)\frac{\begin{array}{c}\Gamma \vdash E : \exists x : A.B \quad \Gamma, w : \exists x : A.B \vdash C : s \\ \Gamma, x : A, y : B \vdash F : [\langle x, y\rangle / w]C\end{array}}{\Gamma \vdash (F \textbf{ where } \langle x, y\rangle = E) : [E/w]C}$$

Wir erweitern auch die β-Reduktion um zwei Regeln für die Summen.

$$G \textbf{ where } \langle x, y\rangle = \langle E, F\rangle \to [E/x, F/y]G$$

$$[\langle x, y\rangle / w]F \textbf{ where } \langle x, y\rangle = E \to [E/w]F$$

Für starke und schwache Summen werden unterschiedliche Abhängigkeiten gebraucht [Jac91]. Daraus leitet Jacobs dann unterschiedliche Anforderungen an eine Semantik ab.

2.7 Typen als Spezifikation

Wie können Typen zur Programmspezifikation benutzet werden? Howard [How80] gibt eine Isomorphie zwischen einem Regelsystem für Typen und einem für Logik an. Betrachten wir hierzu das folgende Regelsystem für eine intuitionistische Logik in natürlicher Deduktion im Sequenzenstil [RS92]:

$$(taut)\frac{}{\Gamma, A \vdash A}$$

$$(\wedge I)\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \quad (\wedge E_l)\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \quad (\wedge E_r)\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}$$

$$(\to I)\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B} \quad (\to E)\frac{\Gamma \vdash A \to B \quad \Gamma \vdash A}{\Gamma \vdash B}$$

$$(\forall I)\frac{\Gamma \vdash A}{\Gamma \vdash \forall x.A} x \notin \mathrm{fv}(\Gamma) \quad (\forall E)\frac{\Gamma \vdash \forall x.A}{\Gamma \vdash [a/x]A}$$

Im Vergleich zu Typsystemen fehlen hier die Terme. Das Einfügen von Termen in diesen Kalkül entspricht dem Mitführen von Beweisen. Der Term

beschreibt ja die Form des Ableitungsbaums. Es gibt einen Term des Typs τ heißt, es gibt eine Ableitung dafür, diese Ableitung entspricht aber gerade einer Herleitung von τ als Aussage. Diese Isomorphie heißt „Curry-Howard-Isomorphie“.

$$\begin{aligned} \text{Typ} &\cong \text{Aussage} \\ \text{Term} &\cong \text{Beweis} \end{aligned}$$

Diese Isomorphie kann auch noch auf Summen und Existenzquantoren übertragen werden. Beweissysteme wie Coq [CCF+95], Lego [LP92] und NuPrl [CAB+86] nutzen diese Analogie aus. In diesem Zusammenhang sind abhängige Typen essentiell, da prädikatenlogische Formeln wie $\forall x.P(x) \rightarrow Q(x)$ Wertabhängigkeiten beinhalten.

Bei der Überprüfung der Typkorrektheit abhängiger Typen, beziehungsweise der Beweiskorrektheit in einem Beweiser, müssen Werte auf Gleichheit untersucht werden. Um die Entscheidbarkeit zu gewährleisten, wird diese Gleichheit als Äquivalenz modulo $=_\beta$ interpretiert. `Vector` $((\lambda x.x)5)$ und `Vector` (5) sind also äquivalent, aber `Vector` $(m+n)$ und `Vector` $(n+m)$ sind verschiedene Typen. Allerdings kann eine Funktion `Vector` $(m+n)$ $\rightarrow$ `Vector` $(n+m)$ leicht aus einem Beweis für $n+m=m+n$ gewonnen werden. Indizierte Typen werden sich genau dadurch unterscheiden, daß $n+m=m+n$ automatisch inferiert wird, dafür aber kompliziertere Zusammenhänge manchmal nicht beschrieben werden können.

2.8 Typrekonstruktion

In einer Programmiersprache wollen wir auf keinen Fall jeden Term mit seinem Typ annotieren, denn das entspräche der Angabe einer ganzen Derivation.

```
((+:Int->Int->Int 3:Int):Int-Int 4:Int):Int
```

Es sollte in der Regel ausreichen, den Typ von Funktionen anzugeben.

2.8.1 Das Hindley-Milner Typsystem

Dieses Typsystem wurde von Hindley [Hin69] und Milner [Mil78] gefunden und später von Damas und Milner ausführlich untersucht [DM82]. Die Besonderheit diese Typsystems ist, daß es völlig ohne Typannotationen auskommt.

Bei den Regeln unterscheiden wir Typen τ

$$\tau = \alpha \mid \iota \mid \tau \to \tau$$

und Typschemata σ

$$\sigma = \tau \mid \forall\alpha.\sigma$$

Es ist eine wesentliche Einschränkung dieses Systems, daß es nur *flache* Typen zuläßt. Quantoren in Typen müssen immer ganz außen auftreten, d.h. Typen oder Typschemata der Form $(\forall\alpha.\alpha \to \alpha) \to \texttt{Int}$ sind, im Gegensatz zu System F, nicht möglich.

Die Termbildungsregeln sind folgende:

$$(var)\frac{(x:\sigma) \in \Gamma}{\Gamma \vdash x : \sigma}$$

$$(\to I)\frac{\Gamma, x:\tau \vdash E : \tau'}{\Gamma \vdash \lambda x.E : \tau \to \tau'}$$

$$(\to E)\frac{\Gamma \vdash E : \tau \to \tau' \quad \Gamma \vdash F : \tau}{\Gamma \vdash EF : \tau'}$$

$$(let)\frac{\Gamma, x:\sigma \vdash E : \tau' \quad \Gamma \vdash F : \sigma}{\Gamma \vdash \textbf{ let } x = F \textbf{ in } E : \tau'}$$

$$(\forall E)\frac{\Gamma \vdash E : \forall\alpha.\sigma}{\Gamma \vdash E : [\tau/\alpha]\sigma}$$

$$(\forall I)\frac{\Gamma \vdash E : \sigma \quad \alpha \notin \mathrm{fv}(\Gamma)}{\Gamma \vdash E : \forall\alpha.\sigma}$$

$(\forall I)$ und $(\forall E)$ lassen den Typ im Term implizit. Er soll von der Typinferenz bestimmt werden. Der Fixpunkt wird über eine Konstante $\textbf{fix} : \forall\alpha.(\alpha \to \alpha) \to \alpha$ und die zusätzliche Reduktionsregel

$$\textbf{fix}f \to f(\textbf{fix}f)$$

realisiert.

Eine wichtige Idee dieses Systems ist es, das *let*, eine Art zweites λ einzuführen. Die beiden folgenden Ausdrücke sind, wenn sie korrekt typisierbar sind, operationell äquivalent.

$$(\lambda x.E)F \equiv \textbf{ let } x = F \textbf{ in } E$$

Der Unterschied ist, daß $\lambda x.E$ auch ohne F ein syntaktisch korrekter Ausdruck ist, ein **let** $x = \ldots$ **in** E gibt es nicht. Dies führt dazu, daß für **let** freizügigere Typregeln einführt werden können. Als Folge davon ist der rechte Term manchmal typisierbar ist, obwohl der linke es nicht ist.

2.8.2 Typinferenz

Um zu zeigen, daß die Typen hier inferierbar sind, wird das folgende, in gewissem Sinne äquivalente, Regelsystem angegeben. ϕ, ϕ' und ϕ'' seien dabei Substitutionen auf Typvariablen, mgu der allgemeinste Unifikator nach Robinson [Rob65] und $\mathrm{gen}(\Gamma, \tau) = \forall\bar{\alpha}.\tau$ wobei $\bar{\alpha} = \mathrm{fv}(\tau) \setminus \mathrm{fv}(\Gamma)$. $\bar{\alpha}$ steht hier und im Folgenden für eine Folge $\alpha_1, \ldots, \alpha_k$ für ein k. Wenn wir $\bar{\alpha}$ wie hier über eine Menge definieren, besteht die Folge aus den Mengenelementen in beliebiger aber fester Reihenfolge.

$$(var^W)\frac{(x : \forall\bar{\alpha}.\tau) \in \Gamma \quad \bar{\beta}\,\mathrm{new}}{\mathrm{id}, \Gamma \vdash^W x : [\bar{\beta}/\bar{\alpha}]\tau}$$

$$(\to I^W)\frac{\phi, (\Gamma, x : \alpha) \vdash^W E : \tau}{\phi, \Gamma \vdash^W \lambda x.E : \phi\alpha \to \tau}$$

$$(\to E^W)\frac{\phi, \Gamma \vdash^W E : \tau' \quad \phi', \phi\Gamma \vdash^W F : \tau \quad \phi'' = \mathrm{mgu}(\phi'\tau', \tau \to \alpha)}{\phi''\phi'\phi, \Gamma \vdash^W EF : \phi''\alpha}$$

$$(let^W)\frac{\phi', (\phi\Gamma, x : \sigma) \vdash^W E : \tau' \quad \phi, \Gamma \vdash^W F : \tau \quad \sigma = \mathrm{gen}(\phi\Gamma, \tau)}{\phi'\phi, \Gamma \vdash^W \textbf{ let } x = F \textbf{ in } E : \tau'}$$

Dieses System kann auch als Algorithmus gelesen werden, wobei $\phi, \Gamma \vdash^W E : \tau$ ein Funktionsaufruf mit Γ und E als Eingabeparameter sowie τ und ϕ als Ausgabeparameter zu lesen ist.

Um die Äquivalenz von $\vdash$ und $\vdash^W$ zu formulieren, brauchen wir noch folgende Definitionen:

Definition 1 (Instanz). *Ein Typ* $[\bar{\tau}/\bar{\alpha}]\tau$ *heißt Instanz von* $\forall\bar{\alpha}.\tau$.

Definition 2 (generische Instanz). $\forall\bar{\beta}.\tau'$ *heißt generische Instanz von* σ, *falls* τ' *Instanz von* σ *ist und* $\bar{\beta} \notin \mathrm{fv}(\sigma)$.

Jetzt formulieren wir die Äquivalenz:

Proposition 1 (Korrektheit(Milner [Mil78])). *Wenn* $\phi, \Gamma \vdash^W E : \tau$ *gilt, dann gilt auch* $\phi\Gamma \vdash E : \tau$.

Proposition 2 (Vollständigkeit(Damas und Milner[DM82])). *Wenn* $\phi\Gamma \vdash E : \forall\bar{\alpha}.\tau$ *gilt, dann gelten auch* $\phi', \Gamma \vdash^W E : \tau'$ *und es gibt ein* ϕ'', *so daß* $\phi = \phi''\phi'$. *Weiter ist* $\forall\bar{\alpha}.\tau$ *generische Instanz von* $\mathrm{gen}(\phi\Gamma, \phi''\tau')$.

Proposition 3 (allgemeinster Typ(Damas und Milner[DM82])). *Wenn* $\phi, \Gamma \vdash^W E : \tau$ *gilt, dann ist* $\mathrm{gen}(\phi\Gamma, \tau)$ *unter allen* σ *mit* $\phi\Gamma \vdash E : \sigma$ *ein allgemeinster Typ, d.h.* σ *ist immer generische Instanz von* $\mathrm{gen}(\phi\Gamma, \tau)$.

2.8.3 Eine allgemeinere Darstellung

In dieser Darstellung ist die Unifikation mit der eigentlichen Typinferenz sehr stark verwoben. Wir sehen hier zwar, daß jedes Typproblem im Prinzip ein Unifikationsproblem ist, uns wird sich aber im weiteren die Frage stellen, welche Erweiterungen des Typsystems welche Erweiterungen der Unifikation verlangen. Daher wird hier eine allgemeinere Darstellung der Typinferenz gegeben. Die vorliegende Arbeit zeigt, daß es die Trennung der eigentlichen Typinferenz von der Unifikation leichter macht, die Fragen

- Kann ich ein bestimmtes Typsystem auf ein bestimmtes Unifikationsproblem zurückführen?
- Kann ein bestimmtes Unifikationsproblem gelöst werden?

unabhängig voneinander zu untersuchen.

Die Unifikationsprobleme, die bei Hindley-Milner auftreten, sind Standardunifikationen mit existentiellen Quantoren:

$$C = (\tau = \tau') \mid \exists\alpha.C \mid C \wedge C'$$

Wir sagen eine Substitution ϕ erfüllt C wenn

- $C = (\tau = \tau')$ und $\phi\tau = \phi\tau'$

- $C = C_1 \wedge C_2$ und ϕ erfüllt C_1 und C_2
- $C = (\exists \alpha.C')$ und ein Typ τ existiert, so daß ϕ auch $[\tau/\alpha]C$ erfüllt.

Wir schreiben $C \vdash^U C'$ für die Konsequenzrelation zwischen solchen Constraints. $C \vdash^U C'$ gilt also, falls alle ϕ, die C erfüllen auch C' erfüllen.

Typschemata können nun auch noch Constraints enthalten:

$$\sigma = \tau \mid \forall \alpha.\sigma \mid C \Rightarrow \sigma$$

Die Typinferenz formulieren wir so:

$$(var^{W'}) \frac{(x : \forall \bar{\alpha}.C \Rightarrow \tau) \in \Gamma}{\exists \bar{\alpha}.C \wedge (\tau = \beta), \Gamma \vdash^{W'} x : \beta} \quad \beta \text{ new}$$

$$(\rightarrow I^{W'}) \frac{C, (\Gamma, x : \gamma) \vdash^{W'} E : \beta}{(\exists \beta, \gamma.C \wedge \alpha = \gamma \rightarrow \beta), \Gamma \vdash^{W'} \lambda x.E : \alpha} \quad \alpha \text{ new}$$

$$(\rightarrow E^{W'}) \frac{C, \Gamma \vdash^{W'} E : \alpha \quad C', \Gamma \vdash^{W'} F : \beta}{(\exists \beta, \alpha.C \wedge C' \wedge \alpha = \beta \rightarrow \gamma), \Gamma \vdash^{W'} EF : \gamma} \quad \gamma \text{ new}$$

$$(let^{W'}) \frac{C, (\Gamma, x : \forall \alpha.C' \Rightarrow \alpha) \vdash^{W'} E : \beta \quad C', \Gamma \vdash^{W'} F : \alpha}{C \wedge \exists \alpha.C', \Gamma \vdash^{W'} \textbf{let } x = F \textbf{ in } E : \beta}$$

Als Verallgemeinerung der Begriffe Instanz und generische Instanz führen wir jetzt die Relation $\sqsubseteq$ ein. Es gilt

$$C'' \vdash^U \forall \bar{\alpha}.C \Rightarrow \tau \sqsubseteq \forall \bar{\beta}.C' \Rightarrow \tau'$$

genau dann, wenn

$$C'' \wedge C \vdash^U \exists \bar{\beta}.C' \wedge \tau = \tau'.$$

$\vdash^U \tau \sqsubseteq \sigma$ war vorher als Instanz und $\vdash^U \sigma \sqsubseteq \sigma'$ als generische Instanz bezeichnet worden. Korrektheit und Vollständigkeit lassen sich nun folgendermaßen formulieren:

Proposition 4 (Korrektheit). *Wenn $C, \Gamma \vdash^{W'} E : \alpha$ und $\vdash^U \phi C$ gelten, dann gilt auch $\phi\Gamma \vdash E : \phi\alpha$.*

Proposition 5 (Vollständigkeit). *Wenn* $\phi\Gamma \vdash E : \forall\bar{\alpha}.\tau$ *gilt, dann gelten auch* $C, \Gamma \vdash^{W'} E : \beta$ *und* $\vdash^{U} \phi[\tau/\beta]C$.

Proposition 6 (allgemeinster Typ). *Wenn* $C, \Gamma \vdash^{W'} E : \alpha$ *gilt, dann ist* $\forall\alpha.C \Rightarrow \alpha$ *ein allgemeinster Typ, d.h. für alle* σ *mit* $\phi\Gamma \vdash E : \sigma$ *gilt* $\phi \vdash^{U} \sigma \sqsubseteq \forall\alpha.C \Rightarrow \alpha$.

Diese Aussagen sind Spezialfälle des später behandelten Systems und bleiben daher hier ohne Beweis.

Neben der Typinferenz interessiert uns natürlich auch, ob in einem typkorrekten Programm nun wirklich keine Typfehler mehr auftreten können. Dazu gab Milner in [Mil78] eine Semantik für Terme an und zeigte, daß der Wert *wrong*, der für Typfehler steht, niemals die Semantik eines typisierbaren Ausdrucks sein kann.

2.8.4 Erweiterungen des Hindley-Milner-Systems

Wir wollen später eine Erweiterung des Hindley-Milner-Systems um Werteparameter in Typen vorstellen. Bevor wir dazu übergehen, betrachten wir einige andere Erweiterungen.

Die wichtigste Erweiterung des Hindley-Milner-Systems sind algebraische Datentypen. Sie waren bereits von Anfang an in ML dabei, wurden bei Betrachtung der Typinferenz oft weggelassen, da sie im Hindley-Milner-System einfach zu integrieren sind. Bereits am Anfang des Kapitels haben wir gesehen, wie damit Summe, Produkt und Rekursion in Typdefinitionen beschrieben werden. In ML werden die Typdefinitionen zusätzlich parameterisiert. Läufer [Läu92] und Perry [Per90] zeigten bereits, wie auch noch existentielle Quantoren in algebraische Typen integriert werden können. Die Parameter, die nur auf der rechten Seite der Definition auftreten, werden automatisch existentiell quantifiziert. Das führt zu folgenden neuen Regeln:

$$(\textbf{data } I)\frac{\textbf{data } \mathtt{T}\bar{\alpha} = \sum_i D_i\bar{\tau}_i}{C, \Gamma \vdash D_i : \forall\bar{\beta}_i.\forall\bar{\alpha}.(\bar{\tau}_i \to \mathtt{T}\bar{\alpha})} \quad \bar{\beta}_i = \mathrm{fv}(\bar{\tau}_i) \setminus \bar{\alpha}$$

$$(\textbf{data } E)\frac{\begin{array}{l}\textbf{data } \mathtt{T}\bar{\alpha} = \sum_i D_i\bar{\tau}_i \\ C, (\Gamma, \bar{y}_i : [\bar{\mu}/\bar{\alpha}, \bar{\nu}_i/\bar{\beta}_i]\bar{\tau}_i) \vdash E_i : \tau \\ C, \Gamma \vdash F : \mathtt{T}\bar{\mu}\end{array}}{C, \Gamma \vdash \textbf{case } F \textbf{ of } \{D_i\bar{y}_i \Rightarrow E_i\} : \tau} \left\{ \begin{array}{l} \bar{\beta}_i = \mathrm{fv}(\bar{\tau}_i) \setminus \bar{\alpha} \\ \bar{\nu}_i \notin \mathrm{fv}(C, \Gamma, \tau) \end{array} \right.$$

Existentielle Datentypen sind für die Darstellung abstrakter Datentypen geeignet [MP88], [CW85], [Läu96]. Für indizierte Typen sind algebraische Typen eine wichtige Voraussetzung. Auch existentielle Quantoren, insbesondere über Indizes, werden für indizierte Typen unabdingbar sein.

Die meisten Erweiterungen bieten einen neuen Typkonstruktor und neue Regeln zur Konstruktion und Destruktion. Manche, wie eben existentielle Quantoren, lassen sich elegant in bisherige Konstrukte integrieren.

Oft könnte in einem expliziten Kalkül statt dem neuen Typkonstruktor auch ein algebraischen Typkonstruktor verwendet werden und statt Konstruktion und Destruktionsoperatoren, algebraische Termkonstruktoren und case eingesetzt werden. Aber die neue syntaktische Konstruktion kann unter Umständen mehr Typen oder mehr Inferenz zulassen. Ein bekanntes Beispiel ist **let** $x = E$ **in** F statt $(\lambda x.E)F$ zu schreiben. Operationell und im expliziten Kalkül ist das kein Unterschied, aber bei der Typinferenz ist manchmal der erste Ausdruck typkorrekt und letzterer nicht. Ein weiterer Grund für manche Erweiterungen ist, daß der Übersetzer die neuen, speziellen Konstrukte besser optimieren kann.

Oft liefert die Analogie des Curry-Howard-Isomorphismus die Intuition für eine Erweiterung. Beispielsweise kann eine Vergleichsoperation auf einem Typ α als Beweis dafür gesehen werden, daß zwei Werte vom Typ α vergleichbar sind. Der von der Typinferenz gefundene Beweis, also die Vergleichsoperation, wird dann zur Laufzeit mit an die Operationen übergeben, die einen solchen Vergleich brauchen. Dort wird dann auf die Vergleichsoperation zugegriffen.

2.8.5 Subtypen

Subtypen haben die Eigenschaft, daß ein Wert vom Typ τ immer da benutzt werden kann, wo ein Wert vom Typ τ' verlangt wird, vorausgesetzt τ ist ein Subtyp von τ'. Wir verwenden eine Einbettungsfunktion als Beweis für die Subtypeigenschaft:

$$(\leq_1)\frac{\Gamma \vdash i : \tau \leq \tau' \quad \Gamma \vdash E : \tau}{\Gamma \vdash (\textbf{coerce}\ i)E : \tau'}$$

$$(\leq_2)\frac{\Gamma \vdash i : \tau_1 \leq \tau_1' \quad \Gamma \vdash i : \tau_2 \leq \tau_2'}{\Gamma \vdash (\textbf{compose}\ ij) : (\tau_1' \to \tau_2) \leq (\tau_1 \to \tau_2')}$$

In einem expliziten Kalkül könnte die Einbettungsfunktion durch eine Funktion $i : \tau \to \tau'$ und (**compose** ij) durch $\lambda f.\lambda x.(j(f(ix)))$ beschrieben werden. Im impliziten Kalkül müssen die (**coerce** i) nicht angegeben werden, sie werden von der Typinferenz bestimmt [FM89], [FM90], [Rey94], [Mit91].

2.8.6 Rekordtypen

Eine weitere Möglichkeit sind Rekordtypen. Sie sind einem Produkt sehr ähnlich. Die Komponenten werden aber nicht durch die Reihenfolge, sondern durch Bezeichner unterschieden. Mathematisch beschreiben wir einen Rekord durch eine partielle Abbildung von Bezeichnern auf Werte und einen Rekortyp durch eine partielle Abbildung von Bezeichnern auf Typen. Ein Beispiel ist

```
{ name="Didier", age=35, sex='m' }.
```

Der Typ des Ausdrucks ist

record(name : `String`, age : `Int`, sex : `Char`).

Rekordtypen sind für die Theorie objektorientierter Sprachen relevant und werden deshalb intensiv untersucht [GM94]. Eine frühe Beschreibung von Rekordtypen findet sich bei Cardelli und Wegner [CW85]. Dort werden drei neue Sprachelemente eingeführt:

- { `l = v` } ist der Rekord mit einem Bezeichner `l`.
- `r || s` ist der Rekord, der als Bezeichner alle Bezeichner aus `r` und `s` hat. Der Wert für einen Bezeichner `l` wird dabei je nachdem wo `l` vorkommt aus `r` oder `s` übernommen. Bezeichner, die in `r` und `s` vorkommen werden entweder verboten (symmetrische Komposition) oder aus `s` übernommen (asymmetrische Komposition).
- `r.l` ist der Wert des Rekords `r` für den Bezeichner `l`.

Manchmal werden aber auch andere Grundoperationen vorgeschlagen:

- `r with l = v` (Extension) ist der Rekord, der für den Bezeichner `l` den Wert `v` hat, ansonsten aber dieselben Werte wie `r`. Hier gibt es eine strikte Version (`r` darf für `l` keinen Wert haben und die freie Version (Der Wert von `l` in `r` ist für das Resultat irrelevant). Das entspricht den zwei möglichen Interpretationen von `r || { l = v }`.

- `r without l` ist der Rekord der für `l` keinen Wert hat, ansonsten aber die Werte von `r`. Diese Operation ist vor allem bei strikter Interpretation der Extension relevant.

Für die Vererbung objektorientierter Sprachen ist es wichtig, daß ein Rekord mit der Bezeichnermenge L auch als ein Rekord mit der Bezeichnermenge $L' \subseteq L$ aufgefaßt werden kann. Zum Beispiel soll die Funktion `older`

```
older x y = x.age > y.age;
```

auf alle Rekords anwendbar sein, die eine Komponente `age` vom Typ `Int` haben.

Eine Möglichkeit dies sicherzustellen ist, einen Rekordtyp $\mathbf{record}(l_1 : \tau_1, \ldots, l_n : \tau_n, \ldots, l_k : \tau_k)$ als Untertyp von $\mathbf{record}(l_1 : \tau_1, \ldots, l_n : \tau_n)$ aufzufassen [CW85]. Der Typ von `older` wäre dann $\mathbf{record}(age : \texttt{Int}) \to \mathbf{record}(age : \texttt{Int}) \to \texttt{Bool}$. Die Untertyprelation stellt sicher, daß Aufrufe mit Rekords mit zusätzlichen Elementen gültig sind.

Eine andere Möglichkeit ist es, eine totale Abbildung von den l_i auf $\{\mathbf{abs}\} \cup \{\mathbf{pres}(\tau) \mid \tau \text{ ein Typ}\}$ zu definieren [Wan87a], [Rém89], [Rém94]. Dann wird die erwünschte Eigenschaft durch Parametrizität sichergestellt. Der Typ von `older` wäre: $\forall\rho.\forall\rho'.\mathbf{record}(age : \mathbf{pres}(\texttt{Int}), \rho) \to \mathbf{record}(age : \mathbf{pres}(\texttt{Int}), \rho') \to \texttt{Bool}$.

Manche Darstellungen verzichten auf `||` und beschränken sich auf die Extension. Dies vereinfacht unter Umständen die Typinferenz, aber auf der anderen Seite entspricht `||` der Mehrfachvererbung im Objektorientierten und wird nur ungern entbehrt [Rém92].

2.8.7 Typdeklarationen

Um echte polymorphe Rekursion zu erlauben, muß eine Fixpunktregel eingeführt werden, die die Deklaration eines Typschemas ermöglicht.

$$(fix)\frac{\Gamma, f : \sigma \vdash E : \sigma}{\Gamma \vdash \mathbf{fix}\ f :: \sigma\ \mathbf{in}\ E}$$

Wenn auf die Deklaration verzichtet wird [MO84], [Myc84], ist die Typinferenz im allgemeinen nur noch semi-entscheidbar [KTU93], [Wel94]. Es gibt allerdings Algorithmen, die in der Praxis gut funktionieren [Hen93b].

Eine weitere Möglichkeit ist es, bei beliebigen Ausdrücken Typdeklarationen zuzulassen [OL96],[Jon97]. Damit können Typschemata an Stellen zugelassen werden, an denen sonst nur Typen erlaubt sind. Daraus resultiert die Möglichkeit, Ausdrücke aus System F zu kodieren.

2.8.8 Überladen

Mehrere Funktionen, die analoges auf verschiedenen Datenstrukturen durchführen, sollen unter einem Namen geführt werden. Das entspricht einer Tabelle von Funktionen, und Definition und Verwendung einer überladenen Funktion entsprechen Tabelleneintrag und Zugriff. Wir können das in einem expliziten Kalkül folgendermaßen beschreiben:

$$(dictI)\frac{\Gamma \vdash E_i : \sigma_i}{\Gamma \vdash (\textbf{overload } \bar{\mu} : \bar{E}) : (\textbf{dict } \bar{\mu} : \bar{\sigma})}$$

$$(dictE)\frac{\Gamma \vdash E : (\textbf{dict } \bar{\mu} : \bar{\sigma})}{\Gamma \vdash (E \textbf{ at } \mu_i) : \sigma_i}$$

Aus welcher syntaktischen Kategorie μ stammt, ist vom Kalkül abhängig.

In ML können diese Tabellen durch Strukturen formuliert werden. Bei Typklassen [WB89], [Jon95] wie in Haskell gibt der Benutzer in Form von Typklassen bestimmte Tabellenformen vor und definiert in Instanzdeklarationen Regeln, nach denen Tabellen zusammengesetzt werden dürfen. Das Typsystem inferiert dann, wo welche Tabellen gebraucht werden, und wie diese zusammengesetzt werden können. Zum Beispiel würde ($\texttt{Eq}\ \alpha$) dem expliziten Typ ($\textbf{dict} == : \alpha \to \alpha \to \texttt{Bool}$) entsprechen. Es wuerde

$$(\textbf{overload } \texttt{IntEq} : ==) : (\textbf{dict } == : \texttt{Int} \to \texttt{Int} \to \texttt{Bool})$$

und

$$== : \forall\alpha.(\textbf{dict } == : \alpha \to \alpha \to \texttt{Bool})$$

gelten.

Läufer [Läu96] verbindet Typklassen und existentielle Typen. Kaes [Kae92] verbindet einen ähnlichen Überladungskalkül mit Subtypen.

Im System O [OWW95], [Weh97] gibt es überladene Funktionen, die einer Typklasse mit einer Operation sehr ähnlich sind. System O ist allerdings

flexibler, was die Verwendung unterschiedlicher Typen für die selbe überladene Funktion angeht. Der Typ einer überladenen Funktion muß nicht vom Benutzer angegeben werden.

Durchschnittstypen [BH90], [Rey94] sind eine weitere Möglichkeit, Überladung zu verwenden. Ein Ausdruck, der den Typ τ, aber auch den Typ τ' hat, hat auch noch den Typ $\tau \cap \tau'$. Es gibt eine Subtyphierarchie, die unter anderem die Eigenschaft $\tau \cap \tau' \leq \tau$ hat. Das Zusammensetzen von Tabellen, bzw. der Zugriff auf Tabellen findet dann in den inferierten Einbettungsfunktionen statt. Dieser Kalkül hat leider die wesentliche Einschränkung, daß ad-hoc Polymorphie nur auf primitiven Funktionen möglich ist. Eine sinnvolle Anwendung sind Variablen in imperativen Sprachen, bei denen zwischen Ausdrücken vom Typ τ und l-Werten vom Typ τ unterschieden wird. l-Werte sind dabei Ausdrücke, denen Werte zugewiesen werden können. Eine Variable, die vom Typ τ deklariert ist, hat beide Typen[Rey94].

2.8.9 Modulsysteme

Bereits in Standard ML gibt es ein Modulsystem [MTH90]. Es erlaubt Typ- und Funktionsdefinitionen in Strukturen zusammenzufassen. Weiter gibt es Signaturen, die Schnittstellen solcher Strukturen beschreiben. Wenn wir unter ausschließlicher Verwendung der Schnittstelle programmieren, können wir später jede Struktur verwenden, die dieser Schnittstelle genügt. Welche Struktur welcher Schnittstelle genügt, stellt die Typinferenz fest.

Es gibt auch Funktoren, die Signaturen auf Signaturen abbilden, sogar Funktoren höherer Ordnung können verwendet werden [HL94]. Jones [Jon96] stellt eine Alternative vor, die auf parametrisierten Signaturen basiert.

Kapitel 3

Indizes, Constraints und Typen

In diesem Kapitel wollen wir die formalen Grundlagen für die Beschreibung des Typsystems und der Typinferenz legen.

Im Typinferenzsystem von Milner [Mil78] wird das Typisierungsproblem in ein äquivalentes Unifikationsproblem transformiert, welches mit dem Algorithmus von Robinson [Rob65] gelöst werden kann. Ähnlich erzeugen wir bei der Typinferenz für indizierte Typen im ersten Schritt ein Unifikationsproblem, oder, anders formuliert, einen Constraint, der dem Typisierungsproblem äquivalent ist, und dessen Lösbarkeit wir anschließend überprüfen müssen. Ein solcher Constraint besteht aus Bedingungen an Typ- und Termvariablen. Um Fragen wie die Äquivalenz zwischen Typisierungsproblem und Constraint oder die Lösbarkeit eines Constraints zu untersuchen, wollen wir nun Syntax und Semantik von solchen Constraints einführen und diskutieren.

Da wir uns nicht auf ein bestimmtes System von Indizes festlegen wollen, stellen wir nur einige Anforderungen, denen ein Indexsystem genügen muß. Anschließend stellen wir einige Beispiele für solche Indexsysteme vor.

Wir erweitern ein allgemeines Indexsystem, um Constraints formulieren zu können, die mächtig genug sind, Typisierungsprobleme darzustellen, für die wir aber auf der anderen Seite einen Lösungsalgorithmus formulieren können.

Die Constraints haben keine Funktionen oder Prädikate höherer Ordnung. Wir werden daher Indexsysteme und Constraints in einem algebraischen Ansatz [Wir90] beschreiben.

Gegenüber anderen Typsystemen ist hier neu, daß neben Typen weitere Größen, nämlich Indizes, in den Typconstraints behandelt werden. Wir brauchen später neben der Gleichheit noch eine zweite Äquivalenzrelation

auf Typen, die Gleichheit unter Nichtbeachtung der Indizes, im folgenden auch als *strukturelle* Gleichheit bezeichnet. Der Grund ist, daß die Muster für die Argumente immer strukturell vom allgemeinsten Typ sein müssen. Da wir dies auch semantisch fassen wollen, benutzen wir für Typconstraints eine Semantik aus zwei Komponenten, wobei die erste Komponente den Constraint exakt betrachtet, die zweite nur strukturell.

Das Kapitel fängt mit einem Beispiel einer Typdefinition an, deren Bedeutung wir informal beschreiben. Wir werden dann die Form eines Indexsystems definieren, Anforderungen an dessen Semantik stellen und daran anschließend ein paar Beispiele für solche Indexsysteme angeben. Wir werden indizierte Typen und Typconstraints einführen und eine Konsequenzrelation auf diesen Typconstraints definieren.

Im letzten Abschnitt geben wir Definitionen für weitere syntaktische Konstrukte wie Typschemata und Kontext an, die wir für die Beschreibung des Typsystems benötigen.

3.1 Beispiel

Bevor wir Indexsysteme, Typen und Typconstraints formal definieren, wollen wir ein Beispiel informal betrachten. Wir beginnen mit einer typischen Definition für Bäume in Haskell oder Gofer:

```
data Tree a =
   Leaf a |
   Branch (Tree a) (Tree a);
```

Wir haben einen Parameter `a`, der für den Typ der Blätter steht. Ein Baum ist entweder ein Blatt mit einer Komponente vom Typ `a` oder ein Verzweigungsknoten mit zwei Teilbäumen.

Eine funktionale Sprache mit indizierten Typen soll nun Typdefinitionen der folgenden Art unterstützen:

```
data Tree a #n ?b =
   Leaf a, n = 1, b |
   Branch (Tree a l bl) (Tree a r br),
      n = l + r, b = (br and bl and (l = r));
```

Dieser Typ hat drei Parameter, einen Typparameter `a` und zwei Indizes, eine ganze Zahl `n` und einen Wahrheitswert `b`. `a` bestimmt den Typ der Blätter,

`n` die Zahl der Blätter und `b` sagt uns, ob der Baum vollständig balanciert ist. Die Sorte der Indizes wird, in der hier verwendeten Syntax, durch vorgestelltes `#`, bzw. `?` ausgedrückt. Auch eine Syntax mit `n:Int` und `b:Bool` wäre denkbar.

Wir gehen hier auch etwas freizügig mit der Unterscheidung zwischen Aussagen und Ausdrücken der Sorte `Bool` um. Wir werden dies später rechtfertigen. Exakter wäre

```
n = 1, b = true
```

und

```
n = l + r,
(b = true) <-> (br = true  and bl = true and (l = r))
```

zu schreiben.

Die Bedeutung der Indizes wird induktiv festgelegt. Die Zahl der Blätter eines Blattes ist 1, dies wird durch `n = 1` ausgedrückt. Die Zahl der Blätter eines zusammengesetzten Baumes berechnet sich aus der Summe der Anzahl der Blätter des rechten und des linken Teilbaums. Dies wird in der Definition durch die Gleichung `n = l + r` festgelegt. Ein Blatt ist immer vollständig balanciert. Ein zusammengesetzter Baum ist vollständig balanciert, wenn dies beide Teilbäume sind und beide Teilbäume gleich groß sind, also dieselbe Anzahl Blätter haben.

Hier haben wir ein Indexsystem verwendet, das Wahrheitswerte und ganze Zahlen als Indizes zuläßt, und neben Gleichheit von Indizes auch bestimmte Verknüpfungen wie `+` und `and` erlaubt. Im nächsten Abschnitt beschreiben wir einen formalen Rahmen für solche Indexsysteme.

3.2 Anforderungen an ein Indexsystem

In einem Indexsystem gibt es noch keine Typen. Es können nur Zusammenhänge zwischen Indizes beschrieben werden. Ein Indexsystem soll durch eine konkrete Algebra für eine Signatur gegeben werden. Im folgenden legen wir fest, wie solche Signaturen und Algebren aussehen dürfen. Wir könnten anstatt einer konkreten Algebra mit bestimmten Eigenschaften auch die Gültigkeit bestimmter Inferenzregeln fordern. Dann wären wir bei der Wahl der Semantik weniger eingeschränkt. Andererseits ist die Einschränkung geringfügig und die Behandlung dafür einfacher.

Definition 3 (Indexsignatur). *Eine* Indexsignatur *ist ein Tupel* $(S, S_q, \texttt{pred}, F)$*, wobei* S *eine Menge von Sorten,* S_q *eine Teilmenge von* S *und* `pred` *ein Element von* S *sind.* F *ist eine Menge von sortierten Operationssymbolen, d.h. jedes Funktionssymbol hat Argumentsorten aus* S^* *und eine Resultatssorte aus* S*.*

S sind die Sorten, die wir später für Termconstraints (Bedingungen an die Indizes) verwenden können, S_q sind Sorten, über die wir quantifizieren dürfen und für die wir eine Gleichheitsoperation haben müssen. Nur Sorten aus S_q dürfen später als Index auftauchen. `pred` ist die Sorte, die für Termconstraints stehen wird.

Definition 4 (Σ-Term). *Sei* $\Sigma = (S, S_q, \texttt{pred}, F)$ *eine Indexsignatur und* V *eine* S*-indizierte Menge von Variablen, dann sei* $T_{\Sigma,s}(V)$ *folgendermaßen definiert:*

$$\frac{x \in V_s}{x \in T_{\Sigma,s}(V)}$$

$$\frac{t_i \in T_{\Sigma,s_i}(V)}{f(t_1, \ldots, t_n) \in T_{\Sigma,s}(V)} \quad f : s_1 \times \ldots \times s_n \to s \in F$$

$$\frac{}{\text{true} \in T_{\Sigma,\texttt{pred}}(V)} \quad \frac{}{\text{indexfalse} \in T_{\Sigma,\texttt{pred}}(V)}$$

$$\frac{t_1 \in T_{\Sigma,\texttt{pred}}(V) \quad t_2 \in T_{\Sigma,\texttt{pred}}(V)}{t_1 \wedge t_2 \in T_{\Sigma,\texttt{pred}}(V)}$$

$$\frac{t_1 \in T_{\Sigma,\texttt{pred}}(V) \quad t_2 \in T_{\Sigma,\texttt{pred}}(V)}{t_1 \vee t_2 \in T_{\Sigma,\texttt{pred}}(V)}$$

$$\frac{t \in T_{\Sigma,\texttt{pred}}(V)}{\neg t \in T_{\Sigma,\texttt{pred}}(V)}$$

$$\frac{t_1 \in T_{\Sigma,q}(V) \quad t_2 \in T_{\Sigma,q}(V)}{t_1 =_q t_2 \in T_{\Sigma,\texttt{pred}}(V)} \quad q \in S_q$$

$$\frac{t \in T_{\Sigma,\mathtt{pred}}(V) \quad \alpha \in V_q}{\exists_q \alpha.t \in T_{\Sigma,\mathtt{pred}}(V \setminus \{\alpha\})} \quad q \in S_q$$

$$\frac{t \in T_{\Sigma,\mathtt{pred}}(V) \quad \alpha \in V_q}{\forall_q \alpha.t \in T_{\Sigma,\mathtt{pred}}(V \setminus \{\alpha\})} \quad q \in S_q$$

Definition 5 (Σ-Termconstraint). *Ein* Σ-Termconstraint *ist ein* Σ*-Term aus* $T_{\Sigma,\mathtt{pred}}(V)$*.*

Termconstraints sind Constraints, in denen noch keine Typen vorkommen. Neben den Termkonstruktoren mehrsortiger Algebren gibt es Boolesche Operatoren und Quantoren. Die Konstante für falsche Aussagen heißt indexfalse, weil es später in der Typconstraintsemantik noch eine weitere Konstante false geben wird. Solange wir über Termconstraints sprechen, entspricht indexfalse jedoch dem üblichen false.

Wir trennen hier nicht zwischen Termen und Formeln, wie das oft üblich ist [Wir90]. Formeln entsprechen bei uns einfach Termen der Sorte `pred`. Dies erlaubt manchmal homogenere Beschreibungen und läßt prinzipiell Formeln als Teile von Termen zu. Die meisten Indexsysteme werden allerdings davon keinen Gebrauch machen.

Definition 6 (Σ-Algebra). *Eine* Σ-Algebra *ist eine* S*-indizierte Menge* A *mit einer Funktion* $A_f : A_{s_1} \times \ldots \times A_{s_n} \to A_s$ *für jeden Operator* $f : s_1 \times \ldots \times s_n \to s$*. Außerdem gelte:*

- $A_{\mathtt{pred}} = \{\text{true}, \text{false}\}$
- $A_s \neq \emptyset$ *für* $s \in S$

Diese Definition entspricht dem Standard für mehrsortige Algebren.

Nun müssen wir die Semantik von Termen und Termconstraints in der Algebra festlegen. Wir brauchen dazu Umgebungen, die die Semantik von freien Variablen festlegen.

Definition 7 (Σ-Umgebung). *Eine* Σ*-Umgebung* ρ *ist nun eine* S*-indizierte Abbildung von* V *nach* A*. Wir schreiben*

$$\rho[a/v](x) = \begin{cases} a, & v = x \\ \rho(x), & \textit{sonst} \end{cases}$$

Jetzt können wir die Semantik für Terme und damit auch für Termconstraints angeben. Die Semantik entspricht wieder dem Standard für mehrsortige Algebren und Formeln auf diesen.

Definition 8 (Σ-Termsemantik). *Für eine gegebene Umgebung ρ und eine Σ-Algebra A definieren wir:*

$$\begin{aligned}
[\![f(t_1,\ldots,t_n)]\!]_{A,\rho} &= A_f([\![t_1]\!]_{A,\rho},\ldots,[\![t_n]\!]_{A,\rho}) \\
[\![\alpha]\!]_{A,\rho} &= \rho(\alpha) \\
[\![\exists_s \alpha.t]\!]_{A,\rho} &= \exists a_s \in A_s.[\![t]\!]_{A,\rho[a_s/\alpha]} \\
[\![\forall_s \alpha.t]\!]_{A,\rho} &= \forall a_s \in A_s.[\![t]\!]_{A,\rho[a_s/\alpha]} \\
[\![t_1 \wedge t_2]\!]_{A,\rho} &= [\![t_1]\!]_{A,\rho} \wedge [\![t_2]\!]_{A,\rho} \\
[\![t_1 \vee t_2]\!]_{A,\rho} &= [\![t_1]\!]_{A,\rho} \vee [\![t_2]\!]_{A,\rho} \\
[\![\neg t]\!]_{A,\rho} &= \neg [\![t]\!]_{A,\rho} \\
[\![t_1 =_q t_2]\!]_{A,\rho} &= ([\![t_1]\!]_{A,\rho} =_{A_q} [\![t_2]\!]_{A,\rho}) \\
[\![\text{indexfalse}]\!]_{A,\rho} &= \text{false} \\
[\![\text{true}]\!]_{A,\rho} &= \text{true}
\end{aligned}$$

Definition 9 (Konsequenzrelation). *Wir können nun eine Konsequenzrelation $\vdash^A$ definieren mit*

$$P \vdash^A R \text{ genau dann, wenn } \forall\rho.([\![P]\!]_{A,\rho} \to [\![R]\!]_{A,\rho})$$

Aus den Definitionen für Termsemantik und Konsequenzrelationen ist unmittelbar ersichtlich, daß wir für diese Konsequenzrelation die üblichen Schlußregeln verwenden dürfen.

Bemerkung 1. *Für $\vdash^A$ gelten die Schlußregeln der klassischen Logik mit Gleichheit.*

Für indizierte Typen sind nur Indexsysteme interessant, für die die Konsequenzrelation $\vdash^A$ entscheidbar ist.

Definition 10 (Indexsystem). *Ein* Indexsystem *ist eine Indexsignatur Σ zusammen mit einer Σ-Algebra A, so daß $\vdash^A P$ für $P \in T_{\Sigma,\mathtt{pred}}(V)$ entscheidbar ist.*

Bevor wir nun von Termconstraints zu Typconstraints übergehen, wollen wir einige Indexsysteme betrachten.

3.3 Einige Indexsysteme

Wenn wir eine funktionale Sprache mit indizierten Typen entwerfen, müssen wir uns für ein Indexsystem entscheiden. Hier wollen wir einige beschreiben.

3.3.1 Polynomiale Gleichungen

Wir können polynomiale Gleichungen folgendermaßen als Indexsystem formalisieren:

Beispiel 1 (Polynomiale Gleichungen).

$$
\begin{array}{lcll}
const_c & : & \texttt{poly}, & \textit{für jedes } c \in \mathbb{C} \\
* & : & \texttt{poly} \times \texttt{poly} \to \texttt{poly} & \\
+ & : & \texttt{poly} \times \texttt{poly} \to \texttt{poly} & \\
- & : & \texttt{poly} \times \texttt{poly} \to \texttt{poly} & \\
S_q & = & \{\texttt{poly}\} & \\
A_{\texttt{poly}} & = & \mathbb{C} &
\end{array}
$$

Die Entscheidbarkeit dieses Systems wird in der Computeralgebra behandelt [Col75]. Leider können mit dem Grundbereich nicht auf $\mathbb{Z}$ ausweichen, da wir sonst kein entscheidbares System mehr haben. Dies liegt an der Unentscheidbarkeit des zehnten Hilbertschen Problems [Mat70]. Wenn wir $\mathbb{R}$ als Grundbereich wählen, können wir sogar noch die Vergleichsoperation hinzunehmen.

Über $\mathbb{C}$ oder $\mathbb{R}$ statt über $\mathbb{Z}$ zu arbeiten, kann ein Problem sein, da beispielsweise $n > 0 \vdash^A n \geq 1$ nicht gilt. Auch Prädikate wie „x ist gerade“ sind mit polynomialen Gleichungen über $\mathbb{C}$ oder $\mathbb{R}$ nicht formulierbar. Daher sind wir mehr an den folgenden Systemen interessiert.

3.3.2 Presburger Arithmetik

Definition 11 (Presburger Arithmetik). *Eine Formel P ist in Presburger Arithmetik, wenn sie entweder*

- *eine lineare Gleichung,*
- *ein linearer Vergleich,*
- *eine logische Verknüpfung zweier Formeln in Presburger Arithmetik oder*

- *eine Quantifizierung über einer Formel in Presburger Arithmetik ist.*

Der Grundbereich für Variablen sind die ganzen Zahlen.

Presburger Arithmetik ist also bereits ziemlich ausdrucksstark, trotzdem ist sie entscheidbar [Coo72]. Für die Datenabhängigkeitsanalyse in Übersetzern wurden bereits leistungsfähige Löser entwickelt [Pug92]. Formal beschreiben wir das System so als Indexsystem:

Beispiel 2 (Presburger Arithmetik). *Wir haben*

$$\begin{array}{lcll} 1 & : & \texttt{linear} & \\ *_n & : & \texttt{linear} \to \texttt{linear} & \text{für jedes } n \in \mathbb{Z} \\ + & : & \texttt{linear} \times \texttt{linear} \to \texttt{linear} & \\ - & : & \texttt{linear} \times \texttt{linear} \to \texttt{linear} & \\ < & : & \texttt{linear} \times \texttt{linear} \to \texttt{pred} & \\ S_q & = & \{\texttt{linear}\} & \\ A_{\texttt{linear}} & = & \mathbb{Z} & \end{array}$$

Wir können Presburger Arithmetik sogar um eine Wahrheitssorte erweitern, ohne die Entscheidbarkeit aufzugeben. Der Vorteil ist, daß wir dann später Typen auch mit Wahrheitswerten parameterisieren können.

Beispiel 3 (Presburger Arithmetik mit Wahrheit). *Wir haben eine weitere Sorte* `bool` *mit*

$$\begin{array}{lcl} \text{inj} & : & \texttt{bool} \to \texttt{pred} \\ \wedge & : & \texttt{bool} \times \texttt{bool} \to \texttt{bool} \\ \vee & : & \texttt{bool} \times \texttt{bool} \to \texttt{bool} \\ \neg & : & \texttt{bool} \to \texttt{bool} \\ S_q & = & \{\texttt{linear}, \texttt{bool}\} \\ A_{\texttt{bool}} & = & \{\text{true}, \text{false}\} \end{array}$$

Wir können die Entscheidbarkeit sehen, indem wir jeder Formel in Presburger Arithmetik mit Wahrheit eine Formel in Presburger Arithmetik zuordnen. Für jede Variable b der Sorte `bool` führen wir eine Variable b' der Sorte `linear` ein. Wir führen auf Termen eine Ersetzung durch, so daß Ausdrücke der Sorte `bool` völlig durch Ausdrücke der Sorte `pred` ersetzt werden.

- $\text{repl}(b) = (b' = 0)$

- $\text{repl}(\text{inj}(B)) = \text{repl}(B)$
- $\text{repl}(\exists b.A) = \exists b'.\text{repl}(A)$
- $\text{repl}(\forall b.A) = \forall b'.\text{repl}(A)$
- $\text{repl}(A = B) = \text{repl}(A) \leftrightarrow \text{repl}(B)$

Auf allen anderen Operatoren arbeitet repl homomorph, zum Beispiel gilt:

$$\text{repl}(A \wedge B) = \text{repl}(A) \wedge \text{repl}(B)$$

Diese Ersetzung erhält die Semantik geschlossener Constraints, erzeugt aber reine Presburger Constraints, die entscheidbar sind. Ein Beispiel für eine Ersetzung:

$$\text{repl}((b \wedge c) = \neg d) = ((b' = 0) \wedge (c' = 0)) \leftrightarrow \neg(d' = 0)$$

Wir können sogar zwischen `pred` und `bool` mischen, wie im Beispiel am Anfang des Kapitels:

$$\begin{aligned}&\text{repl}(b = (br \wedge bl \wedge (l = r))) = \\ &\quad (b' = 0) \leftrightarrow (br' = 0) \wedge (bl' = 0) \wedge (l = r)\end{aligned}$$

Ähnlich können wir auch endliche Bereiche einführen, indem wir $b' = 0$, $b' = 1, \ldots, b' = k$ und $(b' > k \vee b' < 0)$ verwenden.

Auch Listen über endlichen Bereichen sind vorstellbar: $[\text{true}, \text{false}, \text{true}]$ wäre beispielsweise als $2 \cdot 3^2 + 1 \cdot 3 + 2$ kodiert, $>$ entspräche dann länger. $n = \text{len}(l)$ entspräche aber $3^{n-1} \leq l < 3^n$ und das wäre in Presburger Arithmetik nicht kodierbar.

3.4 Indizierte Typen

Um nun Termconstraints zu Typconstraints erweitern, müssen wir Typen formalisieren. Wir nehmen an, daß Typdeklarationen global für das ganze Programm gelten. Wir könnten sie als lokal annehmen [OL96], was den Vorteil hätte, daß Programmtransformationen, die die Einführung neuer Datentypen verlangen, lokal sind, während sie bei uns global sind. Wir wählen den Weg globaler Typdefinitionen, weil sich damit die Darstellung vereinfacht.

Definition 12 (Typsignatur). *Eine* Typsignatur Δ *zu einer gegebenen Indexsignatur* $\Sigma = (S, S_q, \texttt{pred}, F)$ *ist eine Menge von Deklarationen der folgenden Art:*

$$\texttt{data}\quad \texttt{T}\bar{\alpha}(\bar{n} : \bar{s}) = \sum_i D_i\bar{\tau}_i \Leftarrow Q_i, \bar{\beta}_i(\bar{m}_i : \bar{q}_i)$$

Dabei sind die $s_i, q_{i,j}$ *aus* S_q *und die* $\tau_{i,j}$ *aus* $T_{\Sigma,\Delta,\texttt{type}}(\bar{\alpha} \cup \bar{\beta}_i \cup \bar{n} \cup \bar{m}_i)$ *und die* Q_i *aus* $T_{\Sigma,\texttt{pred}}(\bar{n} \cup \bar{m}_i)$.

Die $\bar{\beta}_i$ und $\bar{m}_i$ sind die freien Variablen der $\bar{\tau}_i$. Wir beachten, daß als Sorten hier nur Elemente von S_q zugelassen sind, also nur solche, für die es Gleichheit und Quantoren gibt. Die Definition von $T_{\Sigma,\Delta,\texttt{type}}(V)$ folgt erst anschließend, da sie auf die Definition der Typsignatur bezug nimmt. Dies ist gerechtfertigt, da für sie die Beschaffenheit der $\bar{\tau}_i$ noch nicht relevant ist. Die Baumdeklaration am Anfang des Kapitels sieht in dieser Notation so aus:

$$\begin{array}{l}
\texttt{data}\quad \texttt{T a}(n : \texttt{Int})(b : \texttt{Bool}) \\
\quad = \texttt{Leaf (Tree a l bl) (Tree a r br)} \\
\qquad \Leftarrow (n = 1 \wedge b = \text{true}) \\
\quad l, bl, r, br \\
\quad \mid \texttt{Branch (Tree a l bl) (Tree a r br)} \\
\qquad \Leftarrow (n = l + r \wedge (b = \text{true}) \Leftrightarrow ((br = \text{true}) \wedge (bl = \text{true}) \wedge (l = r)), \\
\quad l, bl, r, br
\end{array}$$

In der Praxis werden wir `#i` und `?b` statt den Sortendeklarationen i : `int` und b : `bool` verwenden. Da wir in konkreten Beispielen meist Presburger Arithmetik mit Wahrheit verwenden, reichen uns diese beiden Sorten. Außerdem lassen wir die explizite Angabe der $\bar{\beta}_i$ und $\bar{m}_i$ weg.

Nun definieren wir, was bei gegebener Index- und Typsignatur als Typ zugelassen ist.

Definition 13 ((Σ, Δ)-Typ). *Sei* V *eine* $S \cup \{\texttt{type}\}$*-indizierte Menge von Variablen. Dann ist die Menge der* (Σ, Δ)-Typen $T_{\Sigma,\Delta,\texttt{type}}(V)$ *induktiv wie folgt definiert:*

$$\frac{\alpha \in V_{\texttt{type}}}{\alpha \in T_{\Sigma,\Delta,\texttt{type}}(V)}$$

$$\frac{t_i \in T_{\Sigma,s_i}(V) \quad \tau_j \in T_{\Sigma,\Delta,\texttt{type}}(V)}{(\texttt{T}\bar{\tau}\bar{t}) \in T_{\Sigma,\Delta,\texttt{type}}(V)} \quad \texttt{data} \quad \texttt{T}\bar{\alpha}(\bar{n} : \bar{s}) = \ldots \in \Delta$$

$$\frac{\tau_1 \in T_{\Sigma,\Delta,\texttt{type}}(V) \quad \tau_2 \in T_{\Sigma,\Delta,\texttt{type}}(V)}{\tau_1 \to \tau_2 \in T_{\Sigma,\Delta,\texttt{type}}(V)}$$

Um die Semantik von Typen beschreiben zu können definieren wir $A_{\texttt{type}}$:

Definition 14 ($A_{\texttt{type}}$). *$A_{\texttt{type}}$ wird induktiv mit den folgenden Regeln gebildet:*

$$\frac{\tau_1 \in A_{\texttt{type}} \quad \tau_2 \in A_{\texttt{type}}}{\tau_1 \to \tau_2 \in A_{\texttt{type}}}$$

$$\frac{\tau_i \in A_{\texttt{type}} \quad a_k \in A_{s_k}}{\texttt{T}\bar{\tau}\bar{a} \in A_{\texttt{type}}} \qquad \texttt{data} \quad \texttt{T}\bar{\alpha}(\bar{n} : \bar{s}) = \ldots \in \Delta$$

Wir schreiben auch hier oft τ. Typen aus $A_{\texttt{type}}$ haben jedoch keine Variablen.

Nun müssen wir auch noch die strukturelle Gleichheit zwischen zwei Typen aus dieser Menge definieren. Die Kongruenz $\cong$ soll dies vermitteln, $\tau \cong \tau'$ heißt: Abgesehen von den Indizes sind τ und τ' gleich.

Definition 15 ($\cong$). *Wir definieren eine Relation auf $A_{\texttt{type}}$:*

$$\frac{\tau_1 \cong \tau_1' \quad \tau_2 \cong \tau_2'}{\tau_1 \to \tau_2 \cong \tau_1' \to \tau_2'} \qquad \frac{\tau_i \cong \tau_i'}{\texttt{T}\bar{\tau}\bar{a} \cong \texttt{T}\bar{\tau}'\bar{a}'}$$

Bemerkung 2. *Es ist leicht zu sehen, daß $\cong$ eine Äquivalenzrelation ist.*

Es gilt also `(Tree Int 5 True)` $\cong$ `(Tree Int 7 False)`, aber trotzdem ist `(Tree Int 5 True)` $\neq$ `(Tree Int 7 False)`.

Da wir bei der Semantik von Typen auch den freien Typvariablen Werte zuordnen müssen, erweitern wir die Umgebungsdefinition.

Definition 16 ((Σ,Δ)-Umgebung). *Eine (Σ,Δ)-*Umgebung *ρ ist eine $S \cup \{\texttt{type}\}$-indizierte Abbildung von V nach A. Die Schreibweise $\rho[a/v]$ übernehmen wir von Σ-Umgebungen.*

Überall, wo bisher Σ-Umgebungen verwendet wurden, können auch (Σ,Δ)-Umgebungen verwendet werden. Nun können wir die Typsemantik angeben. Sie ist mit der Wahl des Indexsystems festgelegt.

Definition 17 ((Σ, Δ)-Typsemantik).

$$\begin{aligned}
[\![t_1 \to t_2]\!]_{A,\rho} &= [\![t_1]\!]_{A,\rho} \to [\![t_2]\!]_{A,\rho} \\
[\![(\mathtt{T}\tau_1, \ldots, \tau_k, t_1, \ldots, t_n)]\!]_{A,\rho} &= \mathtt{T}([\![\tau_1]\!]_{A,\rho}, \ldots, [\![\tau_k]\!]_{A,\rho}, [\![t_1]\!]_{A,\rho}, \ldots, [\![t_n]\!]_{A,\rho}) \\
[\![\alpha]\!]_{A,\rho} &= \rho(\alpha)
\end{aligned}$$

3.5 Typconstraints

Jetzt definieren wir die Typconstraints, die später während der Typinferenz aus einem Typisierungsproblem entstehen, und von der erweiterten Unifikation gelöst werden.

Definition 18 ((Σ, Δ)-Typconstraint). *Sei V eine $S \cup \{\mathtt{type}\}$-indizierte Menge von Variablen.* (Σ, Δ)-Typconstraints $T_{\Sigma,\Delta,\mathtt{prop}}(V)$ *sind induktiv definiert:*

$$\frac{t \in T_{\Sigma,\mathtt{pred}}(V)}{\mathrm{inj}(t) \in T_{\Sigma,\Delta,\mathtt{prop}}(V)}$$

$$\frac{}{\mathrm{false} \in T_{\Sigma,\Delta,\mathtt{prop}}(V)}$$

$$\frac{t_1 \in T_{\Sigma,\Delta,\mathtt{prop}}(V) \quad t_2 \in T_{\Sigma,\Delta,\mathtt{prop}}(V)}{t_1 \wedge t_2 \in T_{\Sigma,\Delta,\mathtt{prop}}(V)}$$

$$\frac{t_1 \in T_{\Sigma,\Delta,\mathtt{prop}}(V) \quad t_2 \in T_{\Sigma,\mathtt{pred}}(V)}{t_2 \to t_1 \in T_{\Sigma,\Delta,\mathtt{prop}}(V)}$$

$$\frac{t_1 \in T_{\Sigma,\Delta,\mathtt{type}}(V) \quad t_2 \in T_{\Sigma,\Delta,\mathtt{type}}(V)}{t_1 =_{\mathtt{type}} t_2 \in T_{\Sigma,\Delta,\mathtt{prop}}(V)}$$

$$\frac{t \in T_{\Sigma,\Delta,\mathtt{prop}}(V) \quad \alpha \in V_q}{\exists_q \alpha.t \in T_{\Sigma,\Delta,\mathtt{prop}}(V \setminus \{\alpha\})} \quad q \in S_q \cup \{\mathtt{type}\}$$

$$\frac{t \in T_{\Sigma,\Delta,\mathtt{prop}}(V) \quad \alpha \in V_q}{\forall_q \alpha.t \in T_{\Sigma,\Delta,\mathtt{prop}}(V \setminus \{\alpha\})} \quad q \in S_q \cup \{\mathtt{type}\}$$

Die Funktion inj ist eine Einbettung von Termconstraints in Typconstraints. Wie wir später sehen werden, sind Termconstraints immer *strukturell* wahr. Dies führt dazu, daß wir true nicht eigens definieren müssen, da inj(true) genau die erwünschte Semantik hat. Andererseits sind inj(indexfalse) und false nicht das gleiche.

Wir haben Gleichheit und alle Quantoren, aber an logischen Verknüpfungen sind nur $\wedge$ und $\rightarrow$ zugelassen, wobei die Implikation dadurch stark eingeschränkt ist, daß auf der linken Seite nur Termconstraints stehen dürfen. Allgemeine Constraints auf der linken Seite der Implikation würden die Unifikation wesentlich komplizierter, wenn nicht unmöglich gestalten.

Die Schwierigkeit bei der folgenden Definition der Typconstraintsemantik ist, daß wir sowohl die exakte Semantik als auch die Semantik unter Nichtberücksichtigung von Indizes angeben wollen. Die Semantik wird also als Paar gegeben, wobei die erste Komponente alles berücksichtigt, die zweite keine Indizes. π_1 und π_2 sind die Projektionen auf die erste bzw. zweite Komponente.

Definition 19 ((Σ, Δ)-Typconstraintsemantik).

$$
\begin{aligned}
[\![\mathrm{inj}(t)]\!]_{A,\rho} &= ([\![t]\!]_{A,\rho}, \mathrm{true}) \\
[\![\mathrm{false}]\!]_{A,\rho} &= (\mathrm{false}, \mathrm{false}) \\
[\![t_1 =_{\mathtt{type}} t_2]\!]_{A,\rho} &= ([\![t_1]\!]_{A,\rho} =_{A_{\mathtt{type}}} [\![t_2]\!]_{A,\rho}, [\![t_1]\!]_{A,\rho} \cong [\![t_2]\!]_{A,\rho}) \\
[\![t_1 \wedge t_2]\!]_{A,\rho} &= (\pi_1([\![t_1]\!]_{A,\rho}) \wedge \pi_1([\![t_2]\!]_{A,\rho}), \pi_2([\![t_1]\!]_{A,\rho}) \wedge \pi_2([\![t_2]\!]_{A,\rho})) \\
[\![t_2 \rightarrow t_1]\!]_{A,\rho} &= (\pi_2([\![t_1]\!]_{A,\rho}) \wedge ([\![t_2]\!]_{A,\rho} \rightarrow \pi_1([\![t_1]\!]_{A,\rho})), \pi_2([\![t_1]\!]_{A,\rho})) \\
[\![\exists_s \alpha.t]\!]_{A,\rho} &= (\exists a_s \in A_s.\pi_1([\![t]\!]_{A,\rho[a_s/\alpha]}), \exists a_s \in A_s.\pi_2([\![t]\!]_{A,\rho[a_s/\alpha]})) \\
[\![\forall_s \alpha.t]\!]_{A,\rho} &= (\forall a_s \in A_s.\pi_1([\![t]\!]_{A,\rho[a_s/\alpha]}), \forall a_s \in A_s.\pi_2([\![t]\!]_{A,\rho[a_s/\alpha]}))
\end{aligned}
$$

Wir beachten insbesondere, daß

$$[\![\mathrm{inj}(\mathrm{true})]\!]_{A,\rho} = (\mathrm{true}, \mathrm{true})$$

aber andererseits

$$[\![\mathrm{inj}(\mathrm{indexfalse})]\!]_{A,\rho} = (\mathrm{false}, \mathrm{true})$$

Weiterhin ist die Semantik von $P \rightarrow C$ so gewählt, daß, auch wenn P falsch ist, C strukturell stimmen muß, damit $P \rightarrow C$ exakt stimmt. Hier folgen zur

Illustration ein paar Beispiele:

$$[\![\forall m, n.m = n \to \texttt{Vector}\ \alpha m = \texttt{Vector}\ \alpha n]\!]_{A,\rho} = (\text{true}, \text{true})$$
$$[\![\forall m, n.m = n + 1 \to \texttt{Vector}\ \alpha m = \texttt{Vector}\ \alpha n]\!]_{A,\rho} = (\text{false}, \text{true})$$
$$[\![\texttt{int} = \texttt{bool}]\!]_{A,\rho} = (\text{false}, \text{false})$$

3.6 Die Konsequenzrelation $\vdash^A$

Damit die Konsequenzrelation gilt verlangen wir, daß die Konsequenzrelation für beide Komponenten gilt. Dann gilt auch für äquivalente Ausdrücke (im Sinne der Konsequenzrelation), daß sie dieselbe Semantik haben.

Definition 20 (Konsequenzrelation). *Für eine Σ-Algebra A wird nun auch eine Konsequenzrelation $\vdash^A$ auf $T_{\Sigma,\Delta,\texttt{prop}}(V)$ definiert:*

$C \vdash^A C'$ *genau dann, wenn* $\forall \rho.\forall i.(\pi_i([\![C]\!]_{A,\rho}) \to \pi_i([\![C']\!]_{A,\rho}))$.

Wir sagen

$C \equiv^A C'$ *genau dann, wenn* $C \vdash^A C'$ *und* $C' \vdash^A C$.

Wir werden oft einfach P statt $\text{inj}(P)$ schreiben. Wir dürfen das wegen folgender Bemerkung.

Bemerkung 3. *Es gilt*

$\text{inj}(P) \vdash^A \text{inj}(R)$ *genau dann, wenn* $P \vdash^A R$

$\text{inj}(\forall n.P) \equiv^A \forall n.\text{inj}(P)$

$\text{inj}(\exists n.P) \equiv^A \exists n.\text{inj}(P)$

$\text{inj}(P \wedge R) \equiv^A \text{inj}(P) \wedge \text{inj}(R)$

$\text{inj}(P \to R) \equiv^A P \to \text{inj}(R)$

Wir stellen nun die grundlegenden Eigenschaften von $\vdash^A$ zusammen. P steht dabei für Termconstraints, A, B, C für Typconstraints.

Proposition 7 (Eigenschaften von $\vdash^A$). *Folgende Regeln sind semantisch korrekt. Wenn also die obere Aussage semantisch korrekt ist, ist das auch die untere.*

$$(taut)\frac{}{C \vdash^A C}$$

$$(\wedge I)\frac{C \vdash^A A \quad C \vdash^A B}{C \vdash^A A \wedge B} \quad (\wedge E_r)\frac{C \vdash^A A \wedge B}{C \vdash^A A} \quad (\wedge E_l)\frac{C \vdash^A A \wedge B}{C \vdash^A B}$$

$$(\to E)\frac{C \vdash^A P \quad C \vdash^A P \to B}{C \vdash^A B} \quad (\to I)\frac{C \wedge P \vdash^A B}{C \vdash^A P \to B}$$

$$(\forall I)\frac{C \vdash^A A}{C \vdash^A \forall\alpha.A}\alpha \notin \mathrm{fv}(C) \quad (\forall E)\frac{C \vdash^A \forall\alpha.A}{C \vdash^A [a/\alpha]A}$$

$$(\exists I)\frac{C \vdash^A [a/\alpha]A}{C \vdash^A \exists\alpha.A} \quad (\exists E)\frac{C \vdash^A \exists\alpha.A \quad C \wedge A \vdash^A B}{C \vdash^A B}\alpha \notin \mathrm{fv}(C, B)$$

$$(=_\tau I_1)\frac{C \vdash^A \tau_1 = \tau_1', \tau_2 = \tau_2'}{C \vdash^A \tau_1 \to \tau_2 = \tau_1' \to \tau_2'} \quad (=_\tau E_1)\frac{C \vdash^A \tau_1 \to \tau_2 = \tau_1' \to \tau_2'}{C \vdash^A \tau_1 = \tau_1', \tau_2 = \tau_2'}$$

$$(=_\tau I_2)\frac{C \vdash^A \bar{\tau} = \bar{\tau}', \bar{t} = \bar{t}'}{C \vdash^A T\bar{\tau}\bar{t} = \bar{\tau}'\bar{t}'} \quad (=_\tau E_2)\frac{C \vdash^A T\bar{\tau}\bar{t} = \bar{\tau}'\bar{t}'}{C \vdash^A \bar{\tau} = \bar{\tau}', \bar{t} = \bar{t}'}$$

$$(=_t refl)\frac{}{C \vdash^A t = t} \quad (=_\tau refl)\frac{}{C \vdash^A \tau = \tau}$$

$$(=_t trans)\frac{C \vdash^A t = t', t' = t''}{C \vdash^A t = t''} \quad (=_\tau trans)\frac{C \vdash^A \tau = \tau', \tau' = \tau''}{C \vdash^A \tau = \tau''}$$

$$(=_t sym)\frac{C \vdash^A t = t'}{C \vdash^A t' = t} \quad (=_\tau sym)\frac{C \vdash^A \tau = \tau'}{C \vdash^A \tau' = \tau}$$

Beweis: Siehe Anhang.

3.7 Strukturelle Gleichheit mit •

Um strukturelle Gleichheit besser handhaben zu können, definieren wir noch einen Operator •, der die Struktur aus Typen und Typconstraints extrahiert. • ist syntaktisch und semantisch definiert, wobei wir erreichen wollen, daß Strukturextraktion und Übergang von der Syntax zur Semantik vertauschbar sind. c_s sei eine beliebige aber feste Konstante der Sorte s.

Definition 21 (•). *Wir führen eine Operation* $\bullet : T_{\Sigma,\Delta,s}(V) \to T_{\Sigma,\Delta,s}(V)$ *für* $s \in \{\mathtt{type}, \mathtt{prop}\}$ *ein:*

$$\begin{aligned}
(t_1 \to t_2)^\bullet &= t_1^\bullet \to t_2^\bullet \\
(\mathtt{T}\tau_1, \ldots, \tau_k, t_1, \ldots, t_n)^\bullet &= \mathtt{T}(\tau_1^\bullet, \ldots, \tau_k^\bullet, c_{s_1}, \ldots, c_{s_n}) \\
\alpha^\bullet &= \alpha \\
\mathrm{inj}(t)^\bullet &= \mathrm{true} \\
(t_1 =_{\mathtt{type}} t_2)^\bullet &= t_1^\bullet =_{\mathtt{type}} t_2^\bullet \\
(C_1 \wedge C_2)^\bullet &= C_1^\bullet \wedge C_2^\bullet \\
(P \to C)^\bullet &= C^\bullet \\
(\exists_s \alpha.C)^\bullet &= \exists_s \alpha.(C^\bullet) \\
(\forall_s \alpha.C)^\bullet &= \forall_s \alpha.(C^\bullet) \\
(\exists_{\mathtt{type}} \alpha.C)^\bullet &= \exists_{\mathtt{type}} \alpha.(C^\bullet) \\
(\forall_{\mathtt{type}} \alpha.C)^\bullet &= \forall_{\mathtt{type}} \alpha.(C^\bullet)
\end{aligned}$$

Wir definieren • jetzt auch noch auf semantische Weise.

Auf $\{\mathrm{true}, \mathrm{false}\} \times \{\mathrm{true}, \mathrm{false}\}$ *definieren wir* • *als* $(x, y)^\bullet = (y, y)$. *Auf* $A_{\mathtt{type}}$ *durch*

$$\begin{aligned}
(\mathtt{T}\tau_1, \ldots, \tau_k, t_1, \ldots, t_n)^\bullet &= \mathtt{T}(\tau_1^\bullet, \ldots, \tau_k^\bullet, A_{c_{s_1}}, \ldots, A_{c_{s_n}}) \\
(t_1 \to t_2)^\bullet &= t_1^\bullet \to t_2^\bullet
\end{aligned}$$

und schließlich noch für Umgebungen

$$\rho_\bullet(\alpha) = \rho(\alpha)^\bullet$$

Ziel ist, daß die Definitionen auf syntaktischer und semantischer Seite übereinstimmen, also $[\![\_^\bullet]\!]_{\rho_\bullet} = [\![\_]\!]^\bullet_\rho$. Dies gilt zumindest für Typgleichungen:

Proposition 8. *Es gilt:*

$$[\![(\tau = \tau')^\bullet]\!]_{\rho_\bullet} = [\![\tau = \tau']\!]^\bullet_\rho.$$

Beweis: Siehe Anhang.

3.8 Typschemata und Kontexte

Wir führen noch ein paar grundlegende syntaktische Konstrukte für die Typinferenz ein. Typschemata bezeichnen parametrisierte Typen. Wie bereits in der alternativen Hindley-Milner Beschreibung im vorigen Kapitel ersetzen wir den Instanzbegriff durch die Relation $\sqsubseteq$ auf Typschemata. Ähnlich wird das auch in anderen Hindley-Milner-Erweiterungen [Jon94],[OSW98] beschrieben.

Definition 22 (Typschema). *Ein* Typschema *ist von der Form* $\forall\bar{\alpha}.\forall\bar{n}.C \Rightarrow \sigma$*, wobei* σ *wiederum ein Typschema oder aber ein Typ ist,* $\bar{\alpha}$ *sind Typvariablen,* $\bar{n}$ *Termvariablen und* C *ist ein Typconstraint.*

Wir unterscheiden nicht zwischen einem Typschema $\forall\bar{\alpha}.\forall\bar{n}.C \Rightarrow \forall\bar{\alpha}'.\forall\bar{n}'.C' \Rightarrow \sigma$ und $\forall\bar{\alpha},\bar{\alpha}'.\forall\bar{n},\bar{n}'.C \wedge C' \Rightarrow \sigma$,wobei $\bar{\alpha}', \bar{n}' \notin \mathrm{fv}(C)$. Wir werden bei allen Definitionen darauf achten, daß für die beiden Formen nichts unterschiedlich definiert wird. Eine Folge dieser Vereinbarung ist, daß sich jedes Typschema in der Form $\forall\bar{\alpha}.\forall\bar{n}.C \Rightarrow \tau$ schreiben läßt.

Wir haben bereits bei der Einführung über Typen gesehen, daß wir die Typkorrektheit eines Ausdrucks nur im Kontext der deklarierten Variablen betrachten können.

Definition 23 (Kontext). *Ein* Kontext *ist eine Folge von Variablendeklarationen* $(x_1 : \sigma_1, \ldots, x_n : \sigma_n)$.

Die Subsumption $\sqsubseteq$ zwischen zwei Typschemata ist nun die Verallgemeinerung des Instanzbegriffs.

Definition 24 (Subsumption). *Es sei*

$$C'' \vdash^A (\forall\bar{\alpha}.\forall\bar{n}.C \Rightarrow \tau) \sqsubseteq (\forall\bar{\beta}.\forall\bar{m}.C' \Rightarrow \tau')$$

genau dann, wenn

$$C'' \wedge C \vdash^A \exists\delta.(\delta = \tau \wedge \exists\bar{\beta}.\exists\bar{m}.(C' \wedge \delta = \tau'))$$

und

$$C'' \vdash^A (x_1 : \sigma_1, \ldots, x_k : \sigma_k) \sqsubseteq (x_1 : \sigma'_1, \ldots, x_k : \sigma'_k)$$

genau dann, wenn

$$\forall i.(C'' \vdash^A \sigma_i \sqsubseteq \sigma'_i).$$

Bemerkung 4. *Wenn $\bar{\beta}$ und $\bar{m}$ nicht frei in τ auftreten, gilt*

$$C'' \vdash^A \forall\bar{\alpha}.\forall\bar{n}.C \Rightarrow \tau \sqsubseteq \forall\bar{\beta}.\forall\bar{m}.C' \Rightarrow \tau'$$

genau dann, wenn

$$C'' \wedge C \vdash^A \exists\bar{\beta}.\exists\bar{m}.(C' \wedge \tau = \tau')$$

gilt.

Die Aussage „τ sei eine Instanz von σ“ läßt sich hier als $\vdash^A \tau \sqsubseteq \sigma$ ausdrücken, die Aussage „σ' sei eine generische Instanz von σ“ als $\vdash^A \sigma' \sqsubseteq \sigma$. Wir werden aber oft die allgemeinere Form mit einem Constraint auf der linken Seite benötigen. Zum Beispiel gilt

$$\beta = \texttt{Int} \vdash^A \texttt{Int} \to \beta \sqsubseteq \forall\alpha.\alpha \to \alpha,$$

wie aus

$$\beta = \texttt{Int} \vdash^A \exists\delta.(\delta = \texttt{Int} \to \beta \wedge \exists\alpha.(\delta = \alpha \to \alpha))$$

zu sehen ist. Es gilt auch

$$\beta = \texttt{Int} \vdash^A \texttt{Int} \to \texttt{Int} \sqsubseteq \forall\alpha.\alpha \to \alpha.$$

Jones beschreibt in seiner Dissertation [Jon94] eine ähnliche Relation auf eingeschränkten Typschemen:

$$(P \mid \sigma) \leq (P' \mid \sigma')$$

Die Definition der Relation beruht auf dem Begriff der generischen Instanz, der bei Jones spezieller gefaßt ist. Dort kann jede Typgleichung $\tau = \tau'$ in eine Menge von Gleichungen $\alpha_i = \tau_i$ umgeformt werden, außerdem können

Constraints keine Typgleichungen beeinhalten und daher sind $C \vdash^A \tau = \tau'$ und $\vdash^A \tau = \tau'$ äquivalent.

Wenn für Jones Relation unser Begriff der generischen Instanz zugrunde gelegt wird, sind folgende Aussagen jeweils äquivalent:

$$(P \mid \sigma) \leq (P' \mid \sigma') \text{ und } \vdash^A P \Rightarrow \sigma \sqsubseteq P' \Rightarrow \sigma'$$

$$C \vdash^A \sigma \sqsubseteq \sigma' \text{ und } (C \mid \sigma) \leq \sigma'$$

Ob man nun $(P \mid \sigma)$ und $\leq$ einführt oder Constraints auf der linken Seite von $\vdash^A \sigma \sqsubseteq \sigma'$ zuläßt, ist eine Geschmacksfrage. Wir sehen C als Bedingung für die Subsumption, nicht als Bestandteil der linken Seite und bevorzugen daher die Schreibweise $C \vdash^A \sigma \sqsubseteq \sigma'$.

Es gelten nun Transitivität und Reflexivität in folgender Weise:

Proposition 9 (Reflexivität der Subsumption). *Es gilt:*

$$C \vdash^A \sigma \sqsubseteq \sigma$$

Beweis: Siehe Anhang.

Proposition 10 (Transitivität der Subsumption). *Aus*

$$C \vdash^A \sigma \sqsubseteq \sigma'$$

und

$$C' \vdash^A \sigma' \sqsubseteq \sigma''$$

folgt

$$C \wedge C' \vdash^A \sigma \sqsubseteq \sigma''$$

Beweis: Siehe Anhang.

Bei Kontexten, die Bedingungen enthalten, stellt sich manchmal die Frage, ob ein Typschema $\forall\bar{\alpha}.\forall\bar{n}.C \Rightarrow \sigma$ unter allen Substitutionen der freien Variablen Instanzen haben kann, oder ob Bedingungen an die freien Variablen gestellt müssen, damit C erfüllbar bleibt. Zum Beispiel hat $(\texttt{Int} = \alpha) \Rightarrow \alpha$ nur Instanzen falls $(\texttt{Int} = \alpha)$. $\text{ext}(\sigma)$ liefert gerade die Bedingungen an die freien Variablen. Diese Notation hilft uns Aussagen über Typschemata und Kontexte zu formulieren, die für den Fall $\text{ext}(\sigma) \neq \text{true}$, bzw. $\text{ext}(\Gamma) \neq \text{true}$ nur eingeschränkt gelten.

Definition 25 (ext). *Wir definieren*

$$\mathrm{ext}(\forall\bar{\alpha}.\forall\bar{n}.C \Rightarrow \sigma) = \exists\bar{\alpha}.\exists\bar{n}.(C \wedge \mathrm{ext}(\sigma))$$

$$\mathrm{ext}(\tau) = \mathrm{true}$$

und für Kontexte

$$\mathrm{ext}(x_1 : \sigma_1, \ldots, x_k : \sigma_k) = \mathrm{ext}(\sigma_1) \wedge \ldots \wedge \mathrm{ext}(\sigma_k).$$

Es gilt jetzt

$$\mathrm{ext}((\mathtt{Int} = \alpha) \Rightarrow \alpha) = (\mathtt{Int} = \alpha).$$

Ein Typschema oder Kontext heißt *konsistent*, falls $\mathrm{ext}(\sigma) = \mathrm{true}$ bzw. $\mathrm{ext}(\Gamma) = \mathrm{true}$[OSW98].

Im folgenden steht τ für Typen, σ für Typschemata, P und R für Termconstraints und C für Typconstraints.

Kapitel 4

Programmiersprache und Typinferenz

Im vorigen Kapitel wurden Indizes und indizierte Typen vorgestellt. In diesem Kapitel wollen wir eine Programmiersprache mit indizierten Typen vorstellen und die Typinferenz für diese Sprache beschreiben. Außerdem zeigen wir, daß zur Laufzeit keine Typfehler mehr auftreten können und machen Aussagen über die Komplexität der Typinferenz.

Wir stellen dabei zuerst die Regeln vor, nach denen der Programmierer sehen kann, ob sein Programm korrekt getypt ist. Im zweiten Schritt geben wir ein deterministisches Regelsystem an, welches zu jedem Programm einen Typconstraint bestimmt, der genau dann erfüllbar ist, wenn das Programm typisierbar ist. Im dritten Schritt geben wir einen Unifikationsalgorithmus an, der entscheidet, ob der Typconstraint erfüllbar ist.

Um die Hypothese zu zeigen, daß keine Laufzeittypfehler auftreten, beweisen wir, daß unter bestimmten Reduktionen die Typisierbarkeit erhalten bleibt. Wenn also ein Evaluator nur diese Reduktionen benützt, kann kein Typfehler passieren.

Die Komplexität der Typinferenz läßt sich schwer exakt behandeln, insbesondere da das Indexsystem variabel ist. Trotzdem können wir für Programme bestimmter Form zeigen, daß der Aufwand zur Typüberprüfung im wesentlichen linear in der Länge des Programms ist. Dies steht im Kontrast zu anderen Aussagen, daß der Aufwand für Programme allgemeiner Form bereits für ML DEXPTIME-vollständig sind [KTU90], [KM89].

4.1 Die Programmiersprache

Als erstes wollen wir eine Grammatik für die Programmiersprache angeben. Es handelt sich hier nur um den Kern der Programmiersprache, im fünften Kapitel wird auch auf ein paar einfache Erweiterungen eingegangen.

$$
\begin{array}{rcl}
E & = & x \\
& | & E_1 E_2 \\
& | & \lambda x.E \\
& | & (\textbf{let } x = E_1 \textbf{ in } E_2) \\
& | & (\textbf{fix } x :: P \Rightarrow \tau \textbf{ in } E) \\
& | & D \\
& | & \textbf{case } F \textbf{ of } \{D_i \bar{y}_i \Rightarrow E_i\}
\end{array}
$$

Eine operationelle Semantik wird im Abschnitt über die operationelle Korrektheit angegeben. Sie entspricht der Semantik geläufiger funktionaler Sprachen.

Oft werden drei Regelsysteme benutzt, um die Typinferenz für eine Programmiersprache zu beschreiben [Jon94],

- ein logisches, das im wesentlichen je eine Regel zur Konstruktion und Destruktion von Typkonstruktionen hat,
- ein syntaktisches, das genau eine Regel pro Termkonstruktion hat und
- ein algorithmisches, welches als attributierte Grammatik gelesen werden kann und bereits die Unifikation beinhaltet.

Wir schlagen hier eine andere Behandlung vor. Wir übernehmen das logische System, geben aber als zweites ein deterministisches System an, das sowohl syntaktisch als auch algorithmisch ist, aber die Unifikation noch nicht beinhaltet. Die Unifikation findet dann in einem weiteren Schritt statt.

Wir halten diese konzeptuelle Trennung von eigentlicher Typinferenz und Unifikation für wichtig, da Typsystem und Unifikation dann getrennt untersucht und erweitert werden können. Es läßt sich damit auch viel leichter feststellen, ob die Unifikation komplizierter wird, wenn das Typsystem an bestimmten Stellen erweitert wird. Eine solche Trennung findet sich auch bei Wand [Wan87b].

Es kann weiter jede Inferenzregel für sich betrachtet werden und es können zum Beispiel verschiedene Regeln für den Fixpunkt gegeneinander abgewägt werden. Für jede Regel wird dann einzeln untersucht, welche Anforderungen sie an die Unifikation stellt.

In einer Implementierung können diese beiden Phasen natürlich miteinander verwoben werden, vielleicht ähnlich wie bei lexikalischer und syntaktischer Analyse in einem Übersetzer.

Wir beginnen mit dem logischen System. Ein Urteil $C, \Gamma \vdash E : \sigma$ ist zu lesen als „Unter der Voraussetzung C und im Kontext der Deklarationen Γ ist E vom Typ σ“. Dabei steht C eigentlich für eine Äquivalenzklasse $[C]_{\equiv^A}$ von Constraints.

Die ersten vier Regeln sind die des Hindley-Milner-Systems mit dem zusätzlichen Constraint, ähnlich wie bei anderen Systemen [Sul96], [OSW98].

$$(var)\frac{(x : \sigma) \in \Gamma}{C, \Gamma \vdash x : \sigma}$$

$$(\to E)\frac{C, \Gamma \vdash E : \tau' \to \tau \quad C, \Gamma \vdash F : \tau'}{C, \Gamma \vdash EF : \tau}$$

$$(\to I)\frac{C, (\Gamma, x : \tau) \vdash E : \tau'}{C, \Gamma \vdash \lambda x.E : \tau \to \tau'}$$

$$(let)\frac{C, \Gamma \vdash E : \sigma \quad C', (\Gamma, x : \sigma) \vdash F : \tau}{C \wedge C', \Gamma \vdash (\mathbf{let}\ x = E\ \mathbf{in}\ F) : \tau}$$

Für rekursive Funktionen brauchen wir den Fixpunktoperator. Dieser kann, weil wir hier Typen deklarieren wollen, nicht als Konstante eingeführt werden. Die Typschemata dieser deklarierten Funktionen sind geschlossen, haben also keine freien Typvariablen.

$$(fix)\frac{C, (\Gamma, x : \sigma) \vdash E : \sigma \quad C \vdash^A \mathrm{ext}(\sigma)}{C, \Gamma \vdash (\mathbf{fix}\ x :: P \Rightarrow \tau\ \mathbf{in}\ E) : \sigma} \begin{cases} \bar{n}, \bar{\alpha} = \mathrm{fv}(P, \tau) \\ \sigma = \forall \bar{\alpha}. \forall \bar{n}. P \Rightarrow \tau \end{cases}$$

Die folgenden beiden Regeln für die Konstruktion und Dekonstruktion von Typschemata sollten nicht in je zwei Regeln zerlegt werden [OSW98].

$$(\forall \Rightarrow I)\frac{C \wedge C', \Gamma \vdash E : \tau}{C \wedge \exists\bar{\alpha}.\exists\bar{n}.C', \Gamma \vdash E : \forall\bar{\alpha}.\forall\bar{n}.C' \Rightarrow \tau} \quad \bar{n}, \bar{\alpha} \notin \mathrm{fv}(C, \Gamma)$$

$$(\forall \Rightarrow E)\frac{C, \Gamma \vdash E : \forall\bar{\alpha}.\forall\bar{n}.C' \Rightarrow \tau \quad C \vdash^A [\bar{\tau}/\bar{\alpha}][\bar{t}/\bar{n}]C'}{C, \Gamma \vdash E : [\bar{\tau}/\bar{\alpha}][\bar{t}/\bar{n}]\tau}$$

Die Existenzregel wird benötigt, um Variablen auf der linken Seite zu eliminieren. Diese Existenzquantifikation in der Voraussetzung entspricht logisch einer Allquantifikation über die ganze Aussage, die für freie Variable implizit gegeben ist.

$$(\exists I)\frac{C, \Gamma \vdash E : \sigma}{\exists\bar{\alpha}.\exists\bar{n}.C, \Gamma \vdash E : \sigma} \quad \bar{n}, \bar{\alpha} \notin \mathrm{fv}(\Gamma, \sigma)$$

Die Gleichheitsregel drückt aus, das Gleiches durch Gleiches ersetzt werden darf. Dazu reicht Gleichheit modulo des Constraints C aus.

$$(=)\frac{C, [\bar{\tau}/\bar{\alpha}][\bar{t}/\bar{n}]\Gamma \vdash E : [\bar{\tau}/\bar{\alpha}][\bar{t}/\bar{n}]\sigma \quad C \vdash^A \bar{t} = \bar{t}', \bar{\tau} = \bar{\tau}'}{C, [\bar{\tau}'/\bar{\alpha}][\bar{t}'/\bar{n}]\Gamma \vdash E : [\bar{\tau}'/\bar{\alpha}][\bar{t}'/\bar{n}]\sigma}$$

Nun kommen noch Konstruktion und Dekonstruktion von algebraischen Datentypen. Die Konstruktion ist einfach: Q_i muß erfüllt sein, um eine Konstruktion zu erlauben.

$$(\textbf{data } I)\frac{\texttt{data T}\bar{\alpha}\bar{n} = \sum_i D_i\bar{\tau}_i \Leftarrow Q_i, \bar{\beta}_i\bar{m}_i \quad C \vdash^A \exists\bar{m}_i.\exists\bar{n}.Q_i}{C, \Gamma \vdash D_i : \forall\bar{m}_i.\forall\bar{n}.\forall\bar{\beta}_i.\forall\bar{\alpha}.Q_i \Rightarrow (\bar{\tau}_i \to T\bar{\alpha}\bar{n})}$$

Die Dekonstruktion von algebraischen Datentypen bringt eine wesentliche Neuerung von indizierten Typen. In den einzelnen Varianten darf die Bedingung Q_i, die ja bei der Konstruktion erfüllt war, verwendet werden, um die Typaussage zu zeigen. Die existentiell quantifizierten Variablen $\bar{\beta}_i$ und $\bar{m}_i$ dürfen in den Varianten auftreten, nicht aber im Resultat.

$$(\textbf{data } E)\frac{\begin{array}{l}\texttt{data T}\bar{\alpha}\bar{n} = \sum_i D_i\bar{\tau}_i \Leftarrow Q_i, \bar{\beta}_i\bar{m}_i \\ (C \wedge \theta Q_i), (\Gamma, \bar{y}_i : \theta\bar{\tau}_i) \vdash E_i : \tau \\ C, \Gamma \vdash F : \theta(T\bar{\alpha}\bar{n})\end{array}}{C, \Gamma \vdash \textbf{case } F \textbf{ of } \{D_i\bar{y}_i \Rightarrow E_i\} : \tau} \left\{\begin{array}{l}\theta = [\bar{m}_i'/\bar{m}_i, \bar{n}'/\bar{n}, \\ \quad \bar{\mu}/\bar{\alpha}, \bar{\nu}_i/\bar{\beta}_i] \\ \bar{m}_i', \bar{\nu}_i \notin \mathrm{fv}(C, \Gamma, \tau)\end{array}\right.$$

Zwei wichtige Aussagen über dieses System wollen wir hier machen. Zum Ersten darf die Bedingung C auf der linken Seite verschärft werden, zum Zweiten können rechts keine unerfüllbaren Constraints auftreten, wenn die linke Seite erfüllbar ist.

Proposition 11 (Verschärfung). *Wenn* $C' \vdash^A C$ *und* $C, \Gamma \vdash E : \sigma$ *gelten, dann gilt auch* $C', \Gamma \vdash E : \sigma$.

Beweis: Siehe Anhang.

Proposition 12 (Konsistenz). *Wenn* $C, \Gamma \vdash E : \sigma$ *gilt, dann gilt auch* $C \wedge \mathrm{ext}(\Gamma) \vdash^A \mathrm{ext}(\sigma)$.

Beweis: Siehe Anhang.

4.2 Typinferenz für indizierte Typen

Nun kommen wir zur Typinferenz. Hier berechnen wir zu einem gegebenen Term mit Kontext einen Typconstraint C, der genau dann erfüllbar ist, wenn es eine Typisierung von E mit diesem Kontext gibt. Wir beschreiben diesen Algorithmus in Form von Urteilen $C, \Gamma \vdash^W E : \alpha$ und lesen das Regelsystem als attributierte Grammatik, wobei E und Γ geerbte und C und die Variable α synthetisierte Attribute sind. Die Methode, ein Regelsystem als attributierte Grammatik zu lesen, geht auf Rémy [Rém89] zurück.

Wir verwenden auf der rechten Seite ausschließlich Variablen, da wir alle Bedingungen an die Typvariablen im Constraint C haben wollen. Dies ist zumindest bei der Fixpunktregel erforderlich, wenn aus C die Implikation $P \to C$ wird.

Ein weiterer Vorteil ist, daß außer der Resultatsvariablen und den freien Variablen im Kontext keine freien Variablen im Constraint vorkommen. Wir können dann auf den Operator $\mathrm{gen}(\Gamma, \tau)$ verzichten. Dieser Operator wird in vielen Inferenzkalkülen für $\forall(\mathrm{fv}(\tau) \setminus \mathrm{fv}(\Gamma)).\tau$ verwendet.

Die ersten drei Regeln sind einfach. Wir beachten bei $(\to E^W)$, daß wir statt $\exists\alpha.\exists\beta.(C \wedge C' \wedge \alpha = \beta \to \delta)$ auch $\exists\beta.[\beta \to \delta/\alpha](C \wedge C')$ hätten schreiben können, was aber wieder algorithmisch und nicht logisch ist.

$$(var^W)\frac{(x : \forall\bar{\alpha}.\forall\bar{n}.C \Rightarrow \tau) \in \Gamma}{\exists\bar{\alpha}.\exists\bar{n}.(C \wedge \delta = \tau), \Gamma \vdash^W x : \delta} \quad \delta \text{ new}$$

$$(\to I^W)\frac{C, (\Gamma, x : \alpha) \vdash^W E : \beta}{\exists\alpha.\exists\beta.(\delta = \alpha \to \beta \wedge C), \Gamma \vdash^W \lambda x.E : \delta} \quad \alpha, \delta \text{ new}$$

$$(\to E^W)\frac{C, \Gamma \vdash^W E : \alpha \quad C', \Gamma \vdash^W F : \beta}{\exists\alpha.\exists\beta.(C \wedge C' \wedge \alpha = \beta \to \delta), \Gamma \vdash^W EF : \delta} \quad \delta \text{ new}$$

Bei der let-Regel wissen wir nun, daß α die einzige freie Variable in C ist, die nicht im Kontext vorkommt, und können $\forall\alpha.C \Rightarrow \alpha$ anstatt $\text{gen}(\Gamma, C \Rightarrow \alpha)$ schreiben.

$$(let^W)\frac{\begin{array}{c} C, \Gamma \vdash^W E : \alpha \\ C', (\Gamma, x : \forall\alpha.C \Rightarrow \alpha) \vdash^W F : \beta \end{array}}{C' \wedge \exists\alpha.C, \Gamma \vdash^W (\textbf{let } x = E \textbf{ in } F) : \beta}$$

Bei der Fixpunktregel zerfällt der Constraint in zwei Teile: Der erste Teil $\exists\bar{\alpha}.\exists\bar{n}.(\delta = \tau \wedge P)$ beschreibt, daß der Ausdruck insgesamt vom Typ $\forall\bar{\alpha}.\forall\bar{n}.P \Rightarrow \tau$ ist, der zweite Teil $\forall\bar{\alpha}.\forall\bar{n}.(P \to \exists\gamma.(C \wedge \tau = \gamma))$ beschreibt die Bedingung, daß der Teilausdruck E vom Typ $\forall\bar{\alpha}.\forall\bar{n}.P \Rightarrow \tau$ ist. Auch dieser zweite Teil hat im allgemeinen freie Variablen, da bei der Verwendung globaler Variablen Bedingungen an deren Typ gestellt werden.

$$(fix^W)\frac{\begin{array}{l} C, (\Gamma, x : \forall\bar{\alpha}.\forall\bar{n}.P \Rightarrow \tau) \vdash^W E : \gamma \\ \exists\bar{\alpha}.\exists\bar{n}.(\delta = \tau \wedge P) \wedge \\ \quad \forall\bar{\alpha}.\forall\bar{n}.(P \to \exists\gamma.(C \wedge \tau = \gamma)), \\ \quad \Gamma \vdash^W (\textbf{fix } x :: P \Rightarrow \tau \textbf{ in } E) : \delta \end{array}}{\left\{ \begin{array}{l} \delta \text{ new} \\ \bar{\alpha}, \bar{n} = \text{fv}(P, \tau) \end{array} \right.}$$

Die ($\textbf{data } I^W$) Regel ist der (var^W) Regel sehr ähnlich.

$$(\textbf{data } I^W)\frac{\texttt{data T}\bar{\alpha}\bar{n} = \sum_i D_i\bar{\tau}_i \Leftarrow Q_i, \bar{\beta}_i\bar{m}_i}{\exists\bar{\alpha}.\exists\bar{\beta}_i.\exists\bar{n}.\exists\bar{m}_i.(\delta = \bar{\tau}_i \to T\bar{\alpha}\bar{n} \wedge Q_i), \Gamma \vdash^W D_i : \delta} \quad \delta \text{ new}$$

Die Implikation mit Q_i auf der linken Seite zu verwenden, um sozusagen die Verwendung von Q_i aus dem Constraint zu streichen, ist eine entscheidende Idee zur Inferenz für indizierte Typen. Die Idee ist, daß $Q_i \to C$ der schwächste Constraint C' ist, für den $C' \wedge Q_i \vdash^A C$ gilt. [1]

[1] Diese Idee fehlte noch in dem Papier [Zen97].

$$(\textbf{data } E^W)\frac{\begin{array}{l}\texttt{data T}\bar{\alpha}\bar{n} = \sum_i D_i\bar{\tau}_i \Leftarrow Q_i, \bar{\beta}_i\bar{m}_i\\ C, \Gamma \vdash^W F : \eta\\ C_i, (\Gamma, \bar{y}_i : \bar{\rho}_i) \vdash^W E_i : \zeta_i\end{array}}{\begin{array}{l}\exists\eta.\exists\bar{\alpha}.\exists\bar{n}.(\eta = T\bar{\alpha}\bar{n} \wedge C \wedge\\ \quad \forall i.\forall\bar{\beta}_i.\forall\bar{m}_i.\exists\zeta_i.\exists\bar{\rho}_i.(Q_i \rightarrow\\ \quad (\bar{\rho}_i = \bar{\tau}_i \wedge C_i \wedge \zeta_i = \delta))),\\ \quad \Gamma \vdash^W \textbf{case } F \textbf{ of } \{D_i\bar{y}_i \Rightarrow E_i\} : \delta\end{array}} \quad \delta, \bar{\rho}_i \text{ new}$$

Die Quantifizierung $\forall i$ ist dabei Metanotation und steht für $(\forall\bar{\beta}_1 \ldots.) \wedge (\forall\bar{\beta}_2 \ldots.) \wedge \ldots.$

Nun kommen wir zu den Aussagen über dieses System. Wir fangen damit an, daß ein allgemeinerer Kontext Γ' keinen stärkeren Constraint verlangt als der Kontext Γ und die Subsumptionsbedingung zusammen.

Proposition 13 (Kontextabschwächung). *Wenn* $C'' \vdash^A \Gamma \sqsubseteq \Gamma'$ *und* $C, \Gamma \vdash^W E : \alpha$ *gelten, dann gilt auch* $C', \Gamma' \vdash^W E : \alpha$*, wobei* $C'' \wedge C \vdash^A C'$*.*

Beweis: Siehe Anhang.

Die Korrektheit sagt uns, daß $\vdash^W$ keine Typisierung inferiert, die im logischen System nicht möglich wäre, die Vollständigkeit sagt, daß wenn im logischen System eine Typisierung existiert, $\vdash^W$ immer eine mindestens so allgemeine findet.

Satz 1 (Korrektheit). *Wenn* $C, \Gamma \vdash^W E : \alpha$ *gilt, dann gilt auch* $C, \Gamma \vdash E : \alpha$*.*

Beweis: Siehe Anhang.

Satz 2 (Vollständigkeit). *Wenn* $C, \Gamma \vdash E : \forall\bar{\alpha}.\forall\bar{n}.C' \Rightarrow \tau$ *gilt, dann existiert eine Ableitung* $C'', \Gamma \vdash^W E : \beta$ *mit* $C \wedge C' \wedge \mathrm{ext}(\Gamma) \vdash^A \exists\beta.(C'' \wedge \beta = \tau)$*.*

Beweis: Siehe Anhang.

4.3 Unifikationsalgorithmus

In diesem Abschnitt stellen wir den Unifikationsalgorithmus vor und zeigen seine Korrektheit.

Da der Algorithmus nur die Erfüllbarkeit eines Typconstraints testen muß, reicht es, geschlossene Typconstraints (ohne freie Variablen) zu betrachten. Wir geben zuerst eine Normalform für Typconstraints an und einen Normalformalgorithmus, der jeden Typconstraint auf Normalform bringt. Wir zeigen, daß sich eine Normalform ohne freie Variablen, die nicht *false* ist, leicht in einen äquivalenten Termconstraint ohne freie Variablen überführen läßt. Da wir aber von Termconstraints Entscheidbarkeit gefordert haben, ist damit das Unifikationsproblem gelöst.

4.3.1 Normalformen

Als erstes beschreiben wir hier die Normalform von Typconstraints.

Definition 26 (Normalform). *Ein Typconstraint C ist* beinahe *in* Normalform*, wenn er* false *ist oder die Form*

$$Q.P \wedge \psi \wedge G$$

hat, wobei Q eine Liste von Quantoren ist, P ein Termconstraint, ψ eine idempotente Substitution $[\bar{\tau}/\bar{\alpha}]$, die wir hier als Menge von aufgelösten Typgleichungen $\alpha_i = \tau_i$ interpretieren, G eine Menge von Variablengleichungen modulo Termconstraint $P \rightarrow (\beta_i = \beta_i')$. Alle Variablen α_i, die in $\mathrm{dom}(\psi)$ *vorkommen, dürfen sonst irgendwo in der Formel auftreten. Außerdem dürfen die τ_i in ψ nie die Form einer einzelnen Variablen annehmen.*

Wenn zusätzlich keine Typvariable universell quantifiziert ist und alle $\alpha \in$ $\mathrm{dom}(\psi)$ *frei in C sind, dann ist C in* Normalform.

Der Algorithmus wird in zwei Transformationen $\mathcal{T}_1$ und $\mathcal{T}_2$ unterteilt. Die erste Transformation bringt den Constraint beinahe in Normalform, der zweite in Normalform.

Es wird eine wiederkehrende Aufgabe sein, eine bedingte Gleichung $P \rightarrow (\tau = \tau')$ in $Q.(\tau'' = \tau''') \wedge P' \wedge (P_i \rightarrow (\alpha_i = \beta_i))$ zu zerlegen. Zum Beispiel machen wir aus

$$(m = n) \rightarrow (\alpha = \texttt{Vector}\ \beta\ n)$$

den Constraint

$$\exists \gamma, k.((\alpha = \texttt{Vector}\ \gamma\ k) \wedge (m = n \rightarrow (\beta = \gamma)) \wedge (m = n \rightarrow n = k)).$$

Diese Aufgabe übernimmt der Algorithmus **M**. Genauer betrachtet entsteht zuerst

$$\exists\gamma, k.((\alpha = \texttt{Vector}\ \gamma\ k) \wedge (m = n \rightarrow (\texttt{Vector}\ \gamma\ k = \texttt{Vector}\ \beta\ n))$$

Die Zerlegung der letzten Komponente übernimmt der Algorithmus **R**. Die Erzeugung des neuen Typs (`Vector` γ k)) aus (`Vector` β n) übernimmt newinst.

Wir stellen den Algorithmus in „bottom-up“ Reihenfolge vor. Dabei machen wir einige Aussagen, die in der Entscheidbarkeit von Typconstraints münden.

4.3.2 Neue Instanzen

Wir wollen öfter einen Typ τ' erzeugen, der die selbe Struktur hat wie ein gegebener Typ τ, aber sonst möglichst allgemein ist. Es sollen nur neue Variablen in τ' vorkommen und keine Variable mehrfach. Dies erreichen wir mit folgender Definition:

Definition 27 (newinst)**.** newinst *ist über folgendes Regelsystem für Typen und Substitutionen definiert.*

$$\frac{}{(\beta, \emptyset, \beta) = \text{newinst}(\alpha)} \quad \text{new } \beta$$

$$\frac{(\bar{\beta}_1, \bar{n}_1, \tau_1') = \text{newinst}(\tau_1) \quad (\bar{\beta}_2, \bar{n}_2, \tau_2') = \text{newinst}(\tau_2)}{(\bar{\beta}_1 \cup \bar{\beta}_2, \bar{n}_1 \cup \bar{n}_2, \tau_1' \rightarrow \tau_2') = \text{newinst}(\tau_1 \rightarrow \tau_2)}$$

$$\frac{(\bar{\beta}_i, \bar{n}_i, \tau_i') = \text{newinst}(\tau_i)}{(\bigcup_i \bar{\beta}_i, \bigcup_i \bar{n}_i, T\bar{\tau}'\bar{n}') = \text{newinst}(T\bar{\tau}\bar{t})} \quad \text{new } \bar{n}'$$

$$\frac{(\bar{\beta}_i, \bar{n}_i, \tau_i') = \text{newinst}(\tau_i)}{(\bigcup_i \bar{\beta}_i, \bigcup_i \bar{n}_i, [\bar{\tau}'/\bar{\alpha}]) = \text{newinst}([\bar{\tau}/\bar{\alpha}])} \quad \text{new } \bar{n}'$$

Die Idee von newinst ist es, einen Constraint $\alpha = \tau$ in eine Bedingung an die Struktur $\alpha = \tau'$ und den nichtstrukturellen Rest $\tau = \tau'$ zu zerlegen. Zum Beispiel ist

$$\text{newinst}(\texttt{T}\beta(\texttt{List}\beta)(m+n)m) = (\langle\mu_1, \mu_2\rangle, \langle\nu_1, \nu_2\rangle, \texttt{T}\mu_1(\texttt{List}\mu_2)\nu_1\nu_2)$$

Der strukturellen Teil

$$\alpha = \mathtt{T}\mu_1(\mathtt{List}\mu_2)\nu_1\nu_2)$$

und der Rest

$$\begin{aligned}(\tau = \tau') &\equiv ((\mathtt{T}\beta(\mathtt{List}\beta)(m+n)m) = \mathtt{T}\mu_1(\mathtt{List}\mu_2)\nu_1\nu_2)\\ &\equiv \beta = \mu_1 = \mu_2 \wedge m + n = \nu_1 \wedge m = \nu_2\end{aligned}$$

sind die Komponenten. Es folgt die zentrale Aussage über newinst.

Lemma 1 (newinst)**.** *Wenn* $C^\bullet \vdash^A \psi$ *und* $(\bar\beta, \bar n, \psi') = \text{newinst}(\psi)$ *gelten, dann gilt auch* $C \vdash^A \exists\bar\beta, \bar n.\psi'$.

Beweis: Siehe Anhang.

4.3.3 Der Algorithmus R

R soll nun den nichtstrukturellen Teil C des Typconstraints in einen Termconstraint P und in bedingte Variablengleichungen G zerlegen, $R(C) = (P, G)$.

Algorithmus 1 (R). **R** *hat als Eingabe einen quantorenfreien Typconstraint.* **R** *arbeitet rekursiv wie folgt:*

- $\mathbf{R}(C \wedge C') = (P \wedge P', G \wedge G')$, *wobei* $(P, G) = \mathbf{R}(C)$ *und* $(P', G') = \mathbf{R}(C')$
- $\mathbf{R}(P \to C) = (P \to P'', (\bigwedge_i (P \wedge P'_i \to (\alpha_i = \tau_i))))$, *wobei* $(P'', (\bigwedge_i P'_i \to (\alpha_i = \tau_i))) = \mathbf{R}(C)$
- $\mathbf{R}(P) = (P, \text{true})$, *wenn* P *bereits die Form eines Termconstraints hat.*
- $\mathbf{R}(\tau_1 \to \tau_2 = \tau'_1 \to \tau'_2) = (P \wedge P', G \wedge G')$, *wobei* $(P, G) = \mathbf{R}(\tau_1 = \tau'_1)$ *und* $(P', G') = \mathbf{R}(\tau_2 = \tau'_2)$
- $\mathbf{R}(\mathtt{T}\tau_1 \ldots \tau_n t_1 \ldots t_k = \mathtt{T}\tau'_1 \ldots \tau'_n t'_1 \ldots t'_k) = ((\bigwedge_i P_i) \wedge (\bigwedge_i t_i = t'_i), (\bigwedge_i G_i))$, *wobei* $(P_i, G_i) = \mathbf{R}(\tau_i = \tau'_i)$
- $\mathbf{R}(\alpha = \tau) = (\text{true}, \alpha = \tau)$
- $\mathbf{R}(\tau = \alpha) = (\text{true}, \alpha = \tau)$

Die beiden letzten Fälle treten nur als $\mathbf{R}(\alpha = \beta)$ auf, wenn das Argument keine Strukturbedingung enthielt. Wir behaupten nun, daß $\mathbf{R}$ die Bedeutung nicht verändert und das Resultat eine bestimmte Struktur hat.

Proposition 14 (Form und Korrektheit von R). *Wenn C ein quantorenfreier Constraint ist, dann ist $\mathbf{R}(C) = (P, G)$ und es gilt $C \equiv^A P \wedge G$, wobei P ein Termconstraint ist und G von der Form $\bigwedge_i (P_i \to \alpha_i = \tau_i)$ ist.*

Wenn weiter $C = \psi' C'$, $\vdash^A \psi C'^\bullet$ und $(\psi', \bar{\beta}, \bar{m}) = \text{newinst}(\psi)$ gilt, dann haben die Typen τ_i die Form einer einzelnen Variablen β_i.

Beweis: Siehe Anhang.

4.3.4 Der Algorithmus M

Der Algorithmus $\mathbf{M}$ führt nun einen öfter wiederkehrenden Teil des Normalformalgorithmus aus. Aus einer Substitution und einer Menge bedingter Typgleichungen wird eine neue Substitution, ein Termconstraint und eine Menge bedingter Variablengleichugen gebildet. Die neu entstehenden Variablen sind existenziell quantifiziert. Die Umformung ist eine Äquivalenzumformung.

Algorithmus 2 (M). *Sei U eine Menge von Typgleichungen und G eine Menge von bedingten Typgleichungen. Wir bilden $\psi'' = \text{mgu}(U^\bullet, G^\bullet)$, den allgemeinsten Unifikator nach Robinson. Aus diesem berechnen wir $(\bar{\beta}, \bar{n}, \psi') := \text{newinst}(\psi'')$. In einem letzten Schritt sei $(P, G') = \mathbf{R}(\psi'(U \wedge G))$. $\mathbf{M}(U, G)$ ist dann $(\psi', P, G', \bar{\beta}, \bar{n})$. Wenn es keinen Unifikator für $U^\bullet$ und $G^\bullet$ gibt, schlägt $\mathbf{M}$ fehl.*

Im Beispiel vorhin wäre

$$U = \text{true}, G = (m = n) \to (\alpha = \texttt{Vector}\ \beta\ n)$$

Es ergibt sich der allgemeinste Unifikator

$$\psi'' = [(\texttt{Vector}\ \beta\ n)/\alpha]$$

und

$$(\gamma, k, [(\texttt{Vector}\ \gamma\ k)/\alpha]) = \text{newinst}(\psi'').$$

Schließlich rufen wir $\mathbf{R}$ auf:

$$\begin{aligned}&(m = n \to n = k, m = n \to \beta = \gamma)\\ &\quad = \mathbf{R}(m = n \to (\texttt{Vector}\ \gamma\ k) = (\texttt{Vector}\ \beta\ n))\end{aligned}$$

Auch für $\mathbf{M}$ haben wir eine Korrektheits- und eine Strukturaussage.

Proposition 15 (Form und Korrektheit M). *Wenn* $\mathbf{M}(U, G) = (\psi', P, G', \bar{\beta}, \bar{n})$ *ist, dann gilt für alle* P' *mit* $\bar{\beta}, \bar{n} \notin \mathrm{fv}(P')$*, daß*

$$\exists\bar{\beta}.\exists\bar{n}.(P' \to P) \wedge \psi' \wedge (P' \to G') \equiv^A P' \to (\psi \wedge G),$$

wobei $\psi' G' = G'$ *und* G' *von der Form* $\bigwedge_i P_i \to (\alpha_i = \beta_i)$ *ist. Wenn* $\mathbf{M}$ *fehlschlägt, ist* $P' \to (U \wedge G) \equiv^A \mathrm{false}$.

Beweis: Siehe Anhang.

4.3.5 Die Transformationen $\mathcal{T}_1$ und $\mathcal{T}_2$

Nun bleibt es noch, die beiden Transformationen $\mathcal{T}_1$ und $\mathcal{T}_2$ mit Hilfe der angegebenen Algorithmen zu beschreiben. Beide definieren wir hier über ein Regelsystem. Die Transformation $\mathcal{T}_1$ besteht im wesentlichen aus dem Herausziehen von Quantoren und dem Anwenden von $\mathbf{M}$. $\mathcal{T}_2$ eliminiert von innen her allquantifizierte Typvariablen und existentiell quantifizierte Variablen aus $\mathrm{dom}(\psi)$.

Algorithmus 3 ($\mathcal{T}_1$). *Die Transformation* $\mathcal{T}_1$*:*

$$\frac{}{\mathcal{T}_1(P) = P}$$

$$\frac{\begin{array}{l}\mathcal{T}_1(C) = Q.P \wedge \psi \wedge G \quad \mathcal{T}_1(C') = Q'.P' \wedge \psi' \wedge G' \\ (\psi'', P'', G'', \bar{\beta}, \bar{n}) = \mathbf{M}(\psi \wedge \psi', G \wedge G')\end{array}}{\mathcal{T}_1(C \wedge C') = Q.Q'.\exists\bar{\beta}.\exists\bar{n}.(P \wedge P' \wedge P'') \wedge \psi'' \wedge G''}$$

$$\frac{\mathcal{T}_1(C) = Q.P \wedge \psi \wedge G \quad (\psi', P'', G'', \bar{\beta}, \bar{n}) = \mathbf{M}(\psi, \mathrm{true})}{\mathcal{T}_1(P' \to C) = Q.\exists\bar{\beta}.\exists\bar{n}.(P' \to (P \wedge P'')) \wedge \psi' \wedge (P' \to (G \wedge G''))}$$

$$\frac{\mathcal{T}_1(C) = Q.P \wedge \psi \wedge G}{\mathcal{T}_1(Q'.C) = Q'.Q.P \wedge \psi \wedge G}$$

Wenn $\mathbf{M}$ *fehlschlägt, ist* $\mathcal{T}_1(C) = \mathrm{false}$.

Proposition 16 (Form und Korrektheit $\mathcal{T}_1$). *Es gilt* $\mathcal{T}_1(C) \equiv^A C$*. Weiter ist* $\mathcal{T}_1(C)$ *beinahe in Normalform.*

Beweis: Siehe Anhang.

Definition 28 (R_G). *Wir definieren die Relation R_G so, daß $\alpha R_G \alpha'$ genau dann gilt, wenn ein $P \to (\alpha = \alpha')$ in G vorkommt. R_G^* bezeichne die reflexive und transitive Hülle.*

Algorithmus 4 ($\mathcal{T}_2$). *Die Transformation $\mathcal{T}_2$:*

$$\frac{\begin{array}{c}\mathcal{T}_2(C) = Q.P \wedge (\psi \wedge \alpha = \tau) \wedge G \\ (\psi', P', G', \bar{\beta}, \bar{n}) = \mathbf{M}([\tau/\alpha], \emptyset)\end{array}}{\mathcal{T}_2(\exists \alpha.C) = \exists \bar{\beta}.\exists \bar{n}.Q.(P \wedge P') \wedge \psi' \wedge (G \wedge G')} \quad \alpha \notin \mathrm{dom}(\psi)$$

$$\frac{\mathcal{T}_2(C) = Q.P \wedge \psi \wedge G}{\mathcal{T}_2(\exists \alpha.C) = \exists \alpha.Q.P \wedge \psi \wedge G} \quad \alpha \notin \mathrm{dom}(\psi)$$

$$\frac{\mathcal{T}_2(C) = Q.P \wedge \psi \wedge G}{\mathcal{T}_2(\forall \alpha.C) = \mathrm{false}} \quad \alpha R_G^* \alpha', \alpha' \in \mathrm{fv}(\psi, \forall \alpha.C)$$

$$\frac{\mathcal{T}_2(C) = Q.P \wedge \psi \wedge (G' \wedge G'')}{\mathcal{T}_2(\forall \alpha.C) = Q.P \wedge \psi \wedge G''} \quad \begin{cases} \mathrm{fv}(G') = \{\alpha' \mid \alpha R_G^* \alpha'\} \\ \mathrm{fv}(G'', \psi, \forall \alpha.C) \cap \mathrm{fv}(G') = \emptyset \end{cases}$$

$$\frac{\mathcal{T}_2(C) = C'}{\mathcal{T}_2(\exists n.C) = \exists n.C'}$$

$$\frac{\mathcal{T}_2(C) = C'}{\mathcal{T}_2(\forall n.C) = \forall n.C'}$$

$$\frac{C = P \wedge \psi \wedge G}{\mathcal{T}_2(C) = P \wedge \psi \wedge G}$$

Wenn $\mathbf{M}$ *fehlschlägt, ist* $\mathcal{T}_2(C) = \mathrm{false}$.

Proposition 17 (Form und Korrektheit $\mathcal{T}_2$). *Es gilt $\mathcal{T}_2(C) \equiv^A C$. Wenn C beinahe in Normalform war, dann ist $\mathcal{T}_2(C)$ in Normalform.*

Beweis: Siehe Anhang.

Bei einem geschlossenen Constraint in Normalform gibt es kein ψ und G ist irrelevant:

Proposition 18 (geschlossene Constraints). *Ein geschlossener Typconstraint in Normalform $Q.P \wedge \psi \wedge G$ ist äquivalent zum Termconstraint $Q.P$*

Beweis: Siehe Anhang.

Hieraus ergibt sich die Entscheidbarkeit:

Satz 3 (Unifikation). *Für einen Typconstraint C ist $\vdash^A C$ entscheidbar.*

Beweis:
Seien $\bar{\alpha}$ und $\bar{n}$ die freien Typ- bzw. Termvariablen von C. $\vdash^A C$ ist genau dann erfüllt, wenn $\vdash^A \forall\bar{\alpha}, \bar{n}.C$ erfüllt ist, dies ist wiederum das gleiche wie $\vdash^A \mathcal{T}_2(\mathcal{T}_1(\forall\bar{\alpha}, \bar{n}.C))$. Das ist aber ein Termconstraint P, dessen Gültigkeit entscheidbar war, oder false. Also ist auch die Gültigkeit von C entscheidbar. □

Korollar 1 (Erfüllbarkeit, Gültigkeit). *Erfüllbarkeit und Gültigkeit von Typconstraints sind entscheidbar.*

4.4 Operationelle Korrektheit

In diesem Abschnitt untersuchen wir die Korrektheit der Typinferenz, also die Frage, ob Programme, die von der Typüberprüfung als korrekt erkannt werden, zur Laufzeit wirklich keine Typfehler produzieren können.

Es gibt dazu zwei Wege. Die erste Möglichkeit ist, eine denotationelle Semantik zu definieren. Wir ordnen dann jedem Ausdruck einen Wert in der Semantik zu und jedem Typ eine Menge von Werten. Dann zeigen wir, daß aus dem Urteil $\bar{x} : \bar{\sigma} \vdash E : \tau$ folgt, daß $[\![E]\!] \in [\![\tau]\!]$ ist, falls $[\![x_i]\!] \in [\![\sigma_i]\!]$ ist [Mil78].

Die Semantik für algebraische Datentypen ist aber sehr kompliziert [Gun92] und wir wählen hier einen einfacheren Weg. Wir werden zeigen, daß die sogenannte Subjektreduktionseigenschaft gilt, das heißt, der Typ eines Ausdrucks bleibt unter Reduktion erhalten. Der Vorteil der ersten Methode wäre, daß bei keiner, im Sinne der denotationellen Semantik korrekten Reduktion, Typfehler auftreten können.

Wir betrachten folgende Reduktionsschritte

$$\begin{array}{rcl}(\lambda x.E)F & \to & [F/x]E \\ (\textbf{fix } f :: \sigma \textbf{ in } E) & \to & [(\textbf{ fix } f :: \sigma \textbf{ in } E)/f]E \\ (\textbf{let } x = E \textbf{ in } F) & \to & [E/x]F \\ \textbf{case } (D\bar{E}) \textbf{ of } \{(D\bar{x}) \Rightarrow E'\} & \to & [\bar{E}/\bar{x}]E'\end{array}$$

Wir legen uns hier nicht auf eine Reduktionsstrategie fest. Jede Strategie, die sich aus diesen Elementarreduktionen zusammensetzen läßt, erfüllt dann die Subjektreduktion.

Das folgende Lemma benötigen wir im Beweis der Subjektreduktionseigenschaft mehrfach. Es ist einer let-Regel sehr ähnlich. Allerdings steht in der Konklusion statt des let eine Substitution, und die Variable x darf an beliebiger Stelle im Kontext auftreten. Die Zusatzbedingung im Antezedent entsteht aus einer Kontextabschwächung.

Mit Γ_x bezeichnen wir den Kontext, in dem gegenüber Γ die Deklaration für x fehlt, der aber ansonsten identisch ist.

Wir schreiben $\to^*$ für die reflexive, transitive und kompatible Hülle von $\to$.

Lemma 2. *Es gilt für* $\bar{\gamma}, \bar{k} \notin \mathrm{fv}(C_5)$

$$\frac{\begin{array}{l}C_0, \Gamma_x \vdash^W F : \nu \quad C_1, \Gamma \vdash^W E : \mu \\ C_5 \wedge C_4 \vdash^A \exists\nu.(C_0 \wedge \nu = \tau) \quad (x : \forall\bar{\gamma}.\forall\bar{k}.C_4 \Rightarrow \tau) \in \Gamma\end{array}}{C_2, \Gamma_x \vdash^W [F/x]E : \mu}$$

und es gilt dann $\exists\nu.C_0 \wedge C_5 \wedge C_1 \vdash^A C_2$.

Beweis: Siehe Anhang.

Wir zeigen die Subjektreduktionseigenschaft im $\vdash^W$ Kalkül, als Korollar gilt sie aber auch im $\vdash$ Kalkül.

Satz 4. *Es gilt*

$$\frac{C, \Gamma \vdash^W F : \alpha \quad F \to^* F'}{C', \Gamma \vdash^W F' : \alpha}$$

wobei $C \vdash C'$.

Beweis: Siehe Anhang.

Korollar 2 (Subjektreduktion). *Es gilt*

$$\frac{C, \Gamma \vdash F : \alpha \quad F \to^* F'}{C \wedge \mathrm{ext}(\Gamma), \Gamma \vdash F' : \alpha}$$

Beweis: Siehe Anhang.

Damit haben wir gezeigt, daß mit den oben angegebenen Reduktionen kein Typfehler passieren kann. Die zusätzliche Bedingung ext(Γ) ist wahrscheinlich unnötig. Sie stört uns aber nicht wirklich, da wir in der Regel nur mit Kontexten arbeiten werden, für die ext(Γ) = true gilt.

4.5 Komplexität der Typüberprüfung

Natürlich ist es schwierig, konkrete Komplexitätsaussagen zu machen, insbesondere, da wir das Indexsystem variabel gehalten haben.

In der Literatur wird bereits für ML gezeigt, daß die Typinferenz DEXPTIME-vollständig ist [KM89], [KTU90]. Die Erfahrung zeigt aber, daß die Typinferenz für praktische ML-Programme linear ist.

Wir zeigen hier, daß für Programme, die eine bestimmte Form haben (im wesentlichen müssen Funktionen von beschränkter Größe sein), der Aufwand für die Typinferenz linear in der Größe des Programms ist.

Satz 5 (Komplexität). *Wenn ein funktionales Programm X von der Form*

$$\begin{array}{l}
\textbf{let } f_1 = (\textbf{fix } f_1 :: P_1 \Rightarrow \tau_1 \textbf{ in } E_1) \textbf{ in} \\
\textbf{let } f_2 = (\textbf{fix } f_2 :: P_1 \Rightarrow \tau_2 \textbf{ in } E_2) \textbf{ in} \\
\vdots \\
\textbf{let } f_m = (\textbf{fix } f_m :: P_1 \Rightarrow \tau_m \textbf{ in } E_m) \textbf{ in} \\
F
\end{array}$$

ist, Γ eine Kontext ohne freie Typvariablen ist, Δ die Menge der Typdeklarationen,

$$|P_i| < b, |\tau_i| < b, |\Delta|_{\mathrm{m}} < b, |E_i| < b, |F| < b, |\Gamma|_{\mathrm{m}} < 3 + 4b$$

und der Aufwand zur Entscheidung von $\vdash^A C$ für $|C| < b'$ durch $B'(b')$ beschränkt ist, dann ist der Aufwand zur Typüberprüfung mit $C, \Gamma \vdash X$:

α durch $B(b)O((|X| + |\Gamma| + |\Delta|)\log(|X| + |\Gamma| + |\Delta|))$ beschränkt, also im wesentlichen linear in der Größe des Programms. Die Größe von C ist durch $B(b)$ beschränkt und damit in $O(1)$ entscheidbar.

Die Nichtlinearität entsteht dabei nur durch die Suche nach Variablen in Γ bzw. Δ. $B(b)$ hängt natürlich nur von b ab und nicht von $|X|$ oder $|\Delta|$. Die Größen sind dabei im wesentlichen als lexikalische Größen definiert, im Beweisanhang sind die Definitionen detailliert angegeben. $|\Gamma|_{\mathrm{m}}$ und $|\Delta|_{\mathrm{m}}$ bezeichnen jeweils die Größe der größten Komponente.

Das der Aufwand zur Entscheidung von $\vdash^A C$ beschränkt ist ergibt sich unmittelbar aus der Entscheidbarkeit und der Tatsache, daß es nur endlich viele wesentlich verschiedene Constraints C beschränkter Größe gibt.

Wir haben damit gezeigt, daß unter gewissen idealisierenden Annahmen der Aufwand für die Typüberprüfung im wesentlichen linear ist, die Frage nach der Praxis läßt sich damit noch nicht beantworten. Wir können jedoch folgern, daß praktische Tests an relativ kleinen Programmen guten Aufschluß über die Proportionalitätskonstante und damit über den praktischen Aufwand der Typüberprüfung geben. Die ersten Ergebnisse einer Prototypimplementierung stimmen zuversichtlich (Das AVL-Beispiel im fünften Kapitel wird in wenigen Sekunden überprüft).

Kapitel 5

Möglichkeiten und Grenzen

In diesem Kapitel wollen wir die Leistungsfähigkeit einer Programmiersprache mit indizierten Typen untersuchen. Wir wollen an Beispielen die Möglichkeiten demonstrieren, die in diesem System stecken, aber auch die Grenzen aufzeigen.

Wir fangen damit an, einige syntaktische Konventionen für die funktionale Sprache zu beschreiben, um die Lesbarkeit der Programmbeispiele zu erhöhen. Wir werden dann die bei Listen gewohnten Komprehensionen auf Vektoren übertragen.

Um Vektor- und Reihungszugriffe zu implementieren, führen wir neue Grunddatentypen `Int'` und `Bool'` ein. Dann betrachten wir Anwendungen der linearen Algebra, wir zeigen die Probleme auf, die dadurch entstehen, daß als Funktionsargumente keine Typschemata zugelassen sind und daß nicht jeder inferierbare Typ in einer Typdeklaration zugelassen ist. Wir zeigen, wie mit Reihungen verfahren werden kann und gehen auf die Möglichkeit, Invarianten von Datenstrukturen im Typ zu kodieren detailliert ein.

Am Ende vergleichen wir indizierte Typen mit allgemeinen abhängigen Typen und den Arbeiten von Xi und Pfenning [XP98], *sized types* von Hughes, Pareto und Sabry [HPS96], *Cayenne* von Augustsson [Aug98]. und *shape types* von Jay und Steckler [JS98]

5.1 Syntaktischer Zucker

Listen sind ein fundamentaler Bestandteil funktionaler Sprachen. Trotzdem kann der Datentyp Liste wie ein Benutzertyp definiert werden.

```
data List a = Nil | Cons a (List a);
```

Weil aber Listen so häufig verwendet werden, gibt es für sie eine spezielle Syntax. Wir verwenden in unserer Sprache `[]` für `Nil`, `[x,y,z]` für `(Cons x (Cons y (Cons z Nil)))` und `[x,y,z:xs]` für `(Cons x (Cons y (Cons z xs)))`. Diese Abkürzungen können sowohl in Ausdrücken als auch in Mustern verwendet werden.

Bei uns wird der Datentyp `Vector` eine ähnlich zentrale Rolle spielen wie Listen in anderen Sprachen.

```
data Vector a #n = Vnil <= (n = 0)
                 | Vcons a (Vector a (n-1)) <= (n > 0);
```

Wir verwenden als Muster entsprechend `<>`, `<x,y,z>` und `<x,y,z:xs>`. 2-Tupel wie

```
data Pair a b = Pair a b;
```

kürzen wir `(a,b)` für `(Pair a b)` ab, entsprechendes gilt für allgemeine n-Tupel. Um das Schreiben von Programmen weiter zu vereinfachen, führen wir noch `if` und das musteranpassende `let` ein. Wir schreiben

```
if b then x else y
```

für

```
case b of { True -> x; False -> y; }
```

und

```
let P = E; in F
```

für

```
let x = E; in case x of { P -> F; }
```

mit einer neuen Variablen `x`.

5.2 Komprehensionen

Die nächste Sonderbehandlung für Listen sind die sogenannten Listenkomprehensionen [Tur82], [Wad87] wie zum Beispiel:

```
[ f x | x <- xs, p x ]
```

Ihre Bedeutung entspricht der einer Mengenbeschreibung

$$\{f(x) \mid x \in xs, p(x)\}$$

Die Ausdrücke der Form `x <- xs` heißen dabei Generatoren, `(p x)` ist ein Filter.

Damit lassen sich nun `quicksort` und `crossproduct` sehr leicht beschreiben. `++` stehe dabei für die Verkettung von Listen.

```
quicksort :: (List Int) -> (List Int);
quicksort [] = [];
quicksort [x:xs] = (quicksort [ y | y <- xs, y <= x]) ++
   [x] ++ (quicksort [ y | y <- xs, y > x]);

crossproduct :: (List a) -> (List b) -> (List (Pair a b));
crossproduct xs ys = [(x,y) | x <- xs, y <- ys];
```

Wir haben hier je ein Beispiel mit Filter und eines mit mehreren Generatoren gewählt. Beides läßt sich, wie wir sehen werden, nicht so ohne weiteres auf Vektoren übertragen.

5.2.1 Zerteiler statt Filter

Betrachten wir zuerst das Beispiel `quicksort` auf Vektoren. Die schwierige Frage ist, wie

```
< y | y <- xs, y <= x >
```

zu interpretieren ist, da die Länge des resultierenden Vektors nicht bekannt ist. Es gibt zwei Möglichkeiten dieses zu beheben. Die erste wäre, eine Liste zurückzugeben. Dies wollen wir aber lieber als Listenkomprehension schreiben und, wie wir später sehen werden, geht das leicht als

```
[ y | y <- (ExList xs), y <= x ]
```

Die zweite Möglichkeit ist es, die Bedingung nicht als Filter, sondern als Zerteiler zu sehen. Wir definieren in einem ersten Ansatz den neuen Datentyp

```
data SplitVector a #n
   = Split (Vector a m) (Vector a (n - m));
```

und interpretieren die Vektorkomprehension

```
< E | x <- xs, P >
```

als einen in zwei Teile geteilten Vektor, wobei in der ersten Hälfte die `x` verwendet werden für die `P` gilt, in der zweiten Hälfte die anderen. Formal ist der Ausdruck gleich

```
svmap (\x -> E) (vsplit  (\x -> P) xs)
```

wobei `vmap` ein `map` für Vektoren ist, und `svmap` eines für `SplitVector`.

```
vmap :: (a -> b) -> (Vector a n) -> (Vector b n);
vmap f <> = <>;
vmap f <x:xs> = <(f x):(vmap f xs)>;

svmap :: (a -> b) -> (SplitVector a n)
      -> (SplitVector b n);
svmap f (Split u v) = (Split (vmap f u) (vmap f v));
```

`vsplit` teilt den Vektor mit Hilfe einer Bedingung in zwei auf und gibt das Ergebnis als `SplitVector`.

```
vsplit :: (a -> Bool) -> (Vector a n)
       -> (SplitVector a n);
vsplit f <> = (Split <> <>);
vsplit f <x:xs> =
   let (Split l r) = (vsplit f xs); in
      if (f x) then (Split <x:l> r) else (Split l <x:r>);
```

Damit können wir quicksort mit Hilfe von `vappend` als

```
vappend :: (Vector a n) -> (Vector a m)
        -> (Vector a (n + m));
vappend <> ys = ys;
vappend <x:xs> ys = <x:(vappend xs ys)>;
```

```
vquicksort :: (Vector Int n) -> (Vector Int n);
vquicksort <> = <> ;
vquicksort <x:xs> =
   let (Split l r) = < y | y <- xs | (ls x y) > in
      vappend (vquicksort l) <x:(vquicksort r)>
```

schreiben. Die Vektorkomprehension ist besser lesbar als

```
vsplit (ls x) xs
```

obwohl letzteres kürzer ist.

5.2.2 Mehrere Generatoren

Ein anderes Problem tritt beim Kreuzprodukt auf. Bei Listenkomprehensionen ist die Länge der entstehenden Liste das Produkt der Längen der ursprünglichen Listen. Damit ist die neue Länge zwar fest, aber mit Presburger-Constraints nicht beschreibbar. Wir können hier nur einen Vektor von Vektoren als Ergebnis geben. Wir ersetzen also einfach

```
< E | x <- xs, y <- ys >
    = < < E | y <- ys > | x <- xs >;
```

Dies unterscheidet sich deutlich von der Semantik der Listenkomprehensionen.

5.2.3 Der allgemeine Fall

Nun haben wir zwei Spezialfälle untersucht. Aber wie behandeln wir den allgemeinen Fall, also mehrere Zerteiler oder mehrere Generatoren im Zusammenhang mit Zerteilern?

Wir verallgemeinern zuerst die Regel für mehrere Generatoren auf zusätzlich vorhandene Zerteiler

```
< E | x <- xs, Z, ..., Z', R >
    = < < E | R > | x <- xs, Z, ..., Z' >
```

wobei `R` mit einem Generator beginnt und die `Z` Zerteiler sind. Damit reduziert sich das Problem auf die Behandlung eines Generators mit mehreren Zerteilern. Wir reduzieren das weiter zu

```
< E | x <- xs, Z, Z', ..., Z'' >
    = < E | x <- < x | x <- xs, Z >, Z',...,Z'' >
```

Dies wäre auch eine gültige Gleichung für Listenkomprehensionen. Vektorkomprehensionen, die mit einem Zerteiler beginnen erlauben wir nicht. Das Problem ist nun, daß wir `Vector` und `SplitVector` durcheinandergeworfen haben und die Ausdrücke nicht mehr typkorrekt sind. Wir müssen also den Ausdruck

```
< E | x <- xs, Z >
```

neu definieren und typisieren. Wir betrachten drei Lösungen. Die erste wäre es, den Datentyp `SplitVector` rekursiv zu definieren

```
data SplitVector a #n
   = Split (SplitVector a m) (SplitVector a (n-m))
   | Single  (Vector a n);
```

und Vektorkomprehensionen auf `SplitVector` zu definieren. Dies hat den Nachteil, daß

```
(Single ys) = < E | x <- xs, P >
```

keinen Typfehler gibt, obwohl klar ist, daß das Ergebnis ein geteilter Vektor ist. Die zweite Alternative wäre es Datentypen

```
data SplitVector a #n
   = Split (Vector a m) (Vector a (n-m));
data SplitVector2 a #n
   = Split2 (SplitVector a m) (SplitVector a (n-m));
```

und so weiter zu definieren und entsprechende `map`-Funktionen auf all diesen Typen zu definieren. Dies hat den Nachteil, daß Vektorkomprehensionen überladen wären oder die Schachtelungstiefe annotiert werden müßte. Überladung gibt es in unserer Sprache nicht, wir bräuchten also eine Spezialbehandlung für Komprehensionen. Andererseits scheint auch eine Kennzeichnung unbequem.

Die dritte und wohl beste Alternative ist, `SplitVector` mit der Schachtelungstiefe zu indizieren, also

```
data SplitVector a #l #n
   = Split (SplitVector a (l-1) m)
           (SplitVector a (l-1) (n-m)) <= (l>0)
   | Single  (Vector a n) <= (l=0);
```

zu setzen. Die Definition von

```
< E | x <- xs, Z >
```

ist nun

```
svmap (\x -> E) (svsplit (\x -> Z) xs)
```

wobei

```
svmap :: (a -> b) -> (SplitVector l n a)
      -> (SplitVector l n b)
svmap f (Split u v) = (Split (svmap f u) (svmap f v));
svmap f (Single v) = (Single (vmap f v));

svsplit :: (SplitVector l n a)
        -> (SplitVector (l+1) n a)
svsplit p (Split u v) = (Split (svsplit f u) (svsplit f v));
svsplit p (Single <>) = (Split (Single <>) (Single <>));
svsplit p (Single <x:xs>) =
   let (Split (Single l) (Single r))
       = svsplit p (Single xs);
   in
       if (p x) then
          (Split (Single <x:l>) (Single r))
       else
          (Split (Single l) (Single <x:r>));
```

ist. Wir brauchen keine Überladung und

```
(Single ys) = < E | x <- xs, P >
```

liefert einen Typfehler. Auch wird die Typinferenz hier in der Regel konkrete Werte für die Schachtelungstiefe `l` inferieren und entsprechende Optimierungen können die Ineffizienzen der zusätzlichen Konstruktoren beseitigen.

Nun schreibt sich Quicksort

```
vquicksort :: (Vector Int n) -> (Vector Int n);
vquicksort <> = <> ;
vquicksort <x:xs> =
   let (Split (Single l) (Single r))
      = < y | y <- (Single xs), (ls x y) > in
      vappend (vquicksort l) <x:(vquicksort r)>
```

Ob sich der zusätzliche Aufwand mit `Single` wirklich lohnt, oder ob mit der einstufigen Version von `SplitVector` und maximal einem Filter pro Generator besser zu arbeiten ist, können erst größere Programme zeigen. Die Alternative mit `Single` kann durch weiteren syntaktischen Zucker sicher auch noch attraktiver gestaltet werden.

5.2.4 Effizienz

Es gibt effiziente Transformationen für Listenkomprehensionen [Wad87]. Diese Transformationen können hier ebenfalls verwendet werden, obwohl die transformierten Programme nicht durch den Typüberprüfer laufen (sie haben Fixpunkte ohne Deklaration). Da sie aber die selben typkorrekten Ergebnisse haben wie die angegebenen Programme, ist dies unerheblich.

5.3 Int', Bool' und Zugriff auf Vektoren

In funktionalen Programmiersprachen sind einige abstrakte Datentypen vordefiniert. Manche davon ließen sich auch vom Programmierenden spezifizieren, wie

```
data Bool = True | False;
```

oder die bereits gezeigte Liste, aber es gibt `if` oder spezielle Muster, wie `[]`, die erwarten, daß die Typen so definiert sind. Manche Typen sind wirklich speziell, wie zum Beispiel

```
data Int = ... | -2 | -1 | 0 | 1 | 2 | ...;
```

und lassen sich auf diese Weise vom Benutzer nicht definieren. Wir geben in unserer Programmiersprache noch weitere derartige Typen vor.

5.3.1 Int' und Bool'

Nehmen wir an, wir wollen formulieren, daß ein Argument eine ganze Zahl zwischen 3 und 5 sein soll. Eine Idee ist, das so zu beschreiben.

```
f :: (2 < i) and (i < 6) => (Int' i) -> ...
```

Dabei ist `(Int' i)` ein Typ der als Wert nur i zuläßt. Die folgende Pseudodefinition ermöglicht das.

```
data Int' #i = ...
             | -2' <= (i = -2)
             | -1' <= (i = -1)
             | 0' <= (i = 0)
             | 1' <= (i = 1)
             | 2' <= (i = 2)
             | ...;
```

Analog definieren wir für Wahrheitswerte.

```
data Bool' ?b = True' <= b
              | False' <= not b;
```

Wir könnten auch auf die Idee kommen, die Wahrheitswerte mit ganzen Zahlen als Indizes zu kodieren:

```
data Bool' #b = True' <= not (b = 0)
              | False' <= b = 0;
```

Aber hier haben wir das Problem, daß `[True', True']` keine homogene Liste mehr ist, es könnte ja `(Bool' 3)` und `(Bool' 4)` sein. Die Kodierung

```
data Bool' #b = True' <= b = 1
              | False' <= b = 0;
```

funktioniert zwar weitgehend, wir müssen aber manchmal

```
case b of { True' -> E; False' -> E; }
```

statt `E` schreiben, um das Wissen, daß b nur 0 oder 1 sein kann, auszunutzen. Diese Unbequemlichkeit bleibt uns mit `Bool'` erspart.

Bei diesen Kodierungen wird ein Zusammenhang zwischen Werten und Indizes hergestellt. Überraschenderweise ist der Zusammenhang zwischen Listen und Vektoren dem zwischen `Int` und `Int'` sehr ähnlich, so daß es vielleicht konsistenter wäre, `List'` statt `Vector` zu schreiben. Wir wollen es aber bei der eingeführten Schreibweise belassen.

5.3.2 Funktionen auf Int' und Bool'

Dieser Zusammenhang läßt sich auch auf Operationen ausdehnen:

```
(+') :: (Int' n) -> (Int' m) -> (Int' (n + m));
less' :: (b = (n < m)) => (Int' n) -> (Int' m) -> (Bool' b);
```

Es können auch nicht-primitive Funktionen geschrieben werden:

```
twice' :: (Int' n) -> (Int' 2*n);
twice' x = x +' x;

half' :: (n > 0) and (2*m-1 < n) and (n < 2*m+2)
      => (Int' n) -> (Int' m);
half' 0' = 0';
half' 1' = 0';
half' n  = (half' (n -' 2')) +' 1';

and' :: (Bool' b) -> (Bool' c) -> (Bool' (b and c));
and' True' x = x;
and' False' x = False';
```

Zu beachten ist, daß dies nur für lineare Funktionen geht, ein `*'` ist also nicht möglich, `twice'` hingegen schon.

Daß `0'` und der Index 0 sich wirklich entsprechen, erfordert einen Meta-Beweis und kann nicht durch den Typüberprüfer bewiesen werden. Aber wenn das für alle Konstanten `i'` so ist, dann zeigt der Typüberprüfer, daß `+'` und `+` sich einander wirklich entsprechen.

5.3.3 Zugriff auf Vektoren

Wir schreiben nun eine Zugriffsfunktion für Vektoren:

```
v_get :: (0 < m) and (m < n + 1)
      => (Vector a n) -> (Int' m) -> a;
v_get <x:xs> 1' = x;
v_get <x:xs> n = v_get xs (n -' 1');
```

Nun ist zum Beispiel in

```
x = v_get <1,2,3> 2'
y = v_get <2,3> 3'
```

der erste Zugriff korrekt, der zweite liefert einen Typfehler. In der Praxis ist oft vorher nicht bekannt, ob der Zugriff auf einen Vektor korrekt wird. Dazu wollen wir eine Funktion

```
v_check_get :: (Vector a n) -> Int -> a;
```

schreiben, die bei falschem Zugriff einen Fehler liefert. Den Kern zur Lösung des Problems liefert die Idee, `Int` nicht als primär aufzufassen, sondern mit Hilfe von `Int'` zu definieren.

```
data Int = (ExInt (Int' n));
...
-2 = (ExInt -2');
-1 = (ExInt -1');
0 = (ExInt 0');
1 = (ExInt 1');
2 = (ExInt 2');
...
```

Genauso für `Bool`:

```
data Bool = (ExBool (Bool' b));
True = (ExBool True');
False = (ExBool False');
```

Hier ist die Analogie zum Zusammenhang zwischen Vektoren und Listen gut erkennbar:

```
data List a = (ExList (Vector a n));
Cons x (ExList xs) = (ExList (Vcons x xs));
Nil = (ExList Vnil);
```

Die Termgleichungen können auch als Muster verwendet werden.

```
len [] = 0;
len [x:xs] = (len xs) + 1;
```

wird zu

```
len (ExList <>) = 0;
len (ExList <x:xs>) = (len (ExList xs)) + 1;
```

eigentlich können wir aber auch die Längenfunktion auf Vektoren benutzen und

```
v_len :: (Vector a n) -> (Int' n);
v_len <> = 0';
v_len <x:xs> = (v_len xs) +' 1';

len (ExList v) = (ExInt (v_len v));
```

schreiben.

Die Ineffizienz, die durch den zusätzlichen Konstruktor zu entstehen scheint, kann vermieden werden. Wenn zu einem Datentyp nur ein strikter Konstruktor mit einem Argument gehört, kann in der Implementierung dieser Konstruktor weggelassen werden. Eine solche Optimierung vermeidet den Overhead der oben angegebenen Konstruktoren. Es entspricht der Intuition, diese Konstruktoren, die einer Typeinbettung entsprechen, strikt zu vereinbaren.

Genauso läßt sich nun ein Quicksort auf Listen mit Hilfe des Quicksort auf Vektoren beschreiben:

```
quicksort (ExList l) = (ExList (vquicksort l));
```

Zur weiteren Vereinfachung führen wir auch hier als syntaktischen Zucker die Syntax

```
if' b then c else d
```

für

```
case b of
  True' -> c
  False' -> d
end
```

ein und schreiben schließlich die überprüfende Zugriffsfunktion auf Vektoren.

```
v_check_get :: (Vector a n) -> Int -> a;
v_check_get v (ExInt m)
      if' (less' m  ((v_len v) +' 1')) then
         error "v_check_get"
      else if' (less' 0' m) then
```

```
            error "v_check_get"
        else
            v_get v m
        end
    end
```

5.4 Lineare Algebra

Wir beginnen mit der Definition des Datentyps für Matrizen:

```
data Matrix a #n #m = Mat (Vector (Vector a n) m);
```

Eine Alternative wäre es, die Dimensionen noch als weitere Argumente des Konstruktors anzugeben. Dies könnte insbesondere für `(Int' n)` sinnvoll sein, da n für $m = 0$ nicht rekonstruiert werden kann.

Eine andere Alternative wäre eine Implementierung mit `(Vector (n * m))`. Die Beschreibung der Multiplikation ist dann mit einem Presburger Indexsystem nicht möglich. Aber eine solche Implementierung ist nicht unbedingt wünschenswert. Schließlich wollen wir ja bei einem Zugriff $1 \leq i \leq n$ und $1 \leq j \leq m$ und nicht $m + 1 \leq i * m + j \leq m * (n + 1)$ wissen.

Wir benutzen im folgenden zwei Funktionen höherer Ordnung, die `foldl` und `zipWith` für Listen entsprechen.

```
vfold :: (a -> b -> b) -> b -> (Vector a n) -> b
vfold f u <> = u;
vfold f u <x:xs> = f (vfold f u xs);

vzipWith :: (a -> b -> c) -> (Vector a n) -> (Vector b n)
         -> (Vector c n)
vzipWith f <> <> = <>
vzipWith f <x:xs> <y:ys> = <(f x y):(vzipWith xs ys)>
```

Bei den Destruktoren geben wir Bedingungen an, so daß sie nie fehlschlagen können.

```
vhead :: (n > 0) => (Vector a n) -> a;
vhead <x:xs> = x;

vtail :: (n > 0) => (Vector a n) -> (Vector a (#n - 1));
vtail <x:xs> = xs;
```

Auch hier ließen sich leicht Versionen `v_check_head` und `v_check_tail` implementieren.

Nun zur linearen Algebra. Für Skalarprodukt und Vektoraddition können wir `vfold` und `vzipWith` sinnvoll einsetzen.

```
sprod :: (Vector Int n) -> (Vector Int n) -> Int;
sprod xs ys = vfold (+) 0 (vzipWith (*) xs ys);

vadd :: (Vector Int n) -> (Vector Int n) -> (Vector Int n);
vadd = vzipWith (+);
```

Für den Nullvektor brauchen wir die Größenangabe.

```
vzero :: (n > 0) => (Int' n) -> (Vector Int n)
vzero 0' = <>;
vzero n  = <0:(vzero (n -' 1'))>
```

`mcons` ist eine Hilfsfunktion zum horizontalen Zusammensetzen von Matrizen.

```
mcons z (Mat x) = (Mat <z:x>);
```

Da `mcons` nicht rekursiv ist, brauchen wir keine Typdeklaration. Nun können wir Addition und Transposition von Matrizen beschreiben.

```
madd :: (Matrix Int n m) -> (Matrix Int n m)
        -> (Matrix Int n m);
madd (Mat <>) (Mat <>) = (Mat <>);
madd (Mat <a:as>) (Mat <b:bs>) =
        (mcons (vadd a b) (madd (Mat as) (Mat bs)));

mtrans :: (m > 0) => (Matrix a n m) -> (Matrix a m n);
mtrans (Mat x)  = case x of
   {
      <<>:xt> -> (Mat <>);
      <<xh1:xh2>:xt> ->
         (mcons (vmap vhead x)
             (mtrans (Mat (vmap vtail x))))
   };
```

Wir stellen fest, daß dieser Algorithmus auf Matrizen mit zweiter Dimension 0 die neue zweite Dimension nicht berechnen kann. Der Datentyp `(Matrix m n)` müßte, wie oben diskutiert, noch eine weitere Komponente `(Int' m)` bekommen, um das zu ermöglichen.

Weiter sind die Muster so gewählt, daß beim Aufruf von `vhead` keine Fehler auftreten können. Hätten wir das nicht, würden wir hier einen Typfehler bekommen.

Bei der Matrixmultiplikation können wir nun die Vektorkomprehension einsetzen.

```
aux_mmult :: (Matrix a n m) -> (Matrix a n k)
          -> (Matrix a m k)
aux_mmult (Mat a) (Mat b) =
   let (Single c)
      =  < (sprod x y) | x <- (Single a), y <- (Single b) >;
   in (Mat c);

mmult :: (Matrix a n m) -> (Matrix a m k)
      -> (Matrix a n k)
mmult a b = aux_mmult (mtrans a) b;
```

Jetzt betrachten wir noch zwei Beispiele, die zu Typfehlern führen. [1]

```
/* ERROR */
madd (Mat <<3, 4>>) (Mat <<7>>);

/* ERROR */
wtrans :: (Matrix a n m) -> (Matrix a m n);
wtrans (Mat x)  = case x of
   {
      <<>:xt> -> (Mat <>);
      <<xh1:xh2>:xt> ->
         (mcons (vmap vhead x) (Mat (vmap vtail x)))
   };
```

[1] Mit dem zweiten Beispiel ist eine persönliche Erfahrung verbunden: Beim Debuggen der Typüberprüfung konnte ich mir lange nicht erklären, warum dieses Beispiel nicht funktionierte, bis ich registrierte, daß der rekursive Aufruf der Transposition fehlte, und die Typfehlermeldung korrekt war.

5.5 Typschemata und Fixpunkt

Bei der oben definierten Funktion `vfold` haben wir zweierlei Probleme. Zuerst einmal ist `Vcons` als Argument nicht zulässig, da es nicht den Typ `a -> b -> b` hat. Dem versierten funktionalen Programmierer ist bestimmt aufgefallen, daß wir nicht

```
mtrans (Mat x) = (Mat (vfold (vzipWith (Vcons)) <> x));
```

geschrieben haben. Dies ist aus dem oben genannten Grund nicht zulässig. Wir können jedoch den Teilausdruck

```
vfold (vzipWith (Vcons)) <>
```

`vvv` nennen und durch ein Rekursion beschreiben. Damit lautet die Transposition:

```
mtrans (Mat x) = let
      vvv :: (Vector (Vector a n) m)
          -> (Vector (Vector a m) n)
      vvv <> = <>;
      vvv <x:xs> = vzipWith (Vcons) x (vvv xs);
   in (Mat (vvv x))
```

Dieses Problem könnten wir lösen, wenn wir Typschemata als Funktionsargumente zuließen [OL96]. Aber selbst wenn das ginge, hätten wir Schwierigkeiten, das allgemeinste Typschema für das Argument zu finden. Wir würden gerne folgende Typschemata unter einen Hut bringen (alle wären dynamisch gültige Argumente für `vfold`):

```
a -> b -> b
a -> (Vector b n) -> (Vector b (n + 1))
a -> (Vector b n) -> (Vector b (n + m))
```

Für die ersten beiden gibt kein gemeinsames Typschema, das dritte würde als Resultattyp von `vfold` einen `(Vector b (n * m))` liefern, den wir nicht beschreiben können.

Als weiteres Beispiel betrachten wir

```
data M #m #n a = D (Vector (Vector a n) m)
               | C (Vector (Vector a n) m) <= n = m;
```

```
f (D x) u v = True;
f (C x) u v = (u v);
```

Es stellt sich die Frage nach dem Typschema für `f`. Folgende Deklarationen sind beide typkorrekt,

```
f :: (M m n) -> (a -> Bool) -> a -> Bool;
f :: (M m n) -> ((Vector a n) -> Bool)
             -> (Vector a m) -> Bool;
```

aber je nach Deklaration ist nur je einer der beiden folgenden Aufrufe korrekt:

```
f (D <<>>) odd 3;

g :: (Vector Int 1) -> Bool;
f (D <<>>) g <>;
```

Es gibt ein allgemeinstes Typschema, welches beide Aufrufe zuließe, nämlich:

```
f :: (m = n -> a = b) => (M m n)
  -> (a -> Bool) -> b -> Bool;
```

Aber diese Typschema ist in Deklarationen nicht zugelassen, da es die bedingte Typgleichung `(m = n -> a = b)` enthält.

In diesem Fall ist `f` aber glücklicherweise nicht rekursiv und eine Definition ohne Deklaration reicht aus. Wenn wir den Typ nicht deklarieren, wird für `f` das allgemeinste Typschema gefunden und beide Aufrufe sind typkorrekt.[2] Das Typschema

```
f :: (m = n -> (a = (b -> Bool))) => (M m n)
  -> a -> b -> Bool;
```

wäre in diesem Fall äquivalent, da wir verlangen, daß `(a -> Bool)` und `b` strukturell äquivalent sind. Der Aufruf

```
f (D <<>>) 3 5;
```

ist also in jedem Fall ein Typfehler.

[2]In einer früheren, unvollständigen Version der Typinferenz konnte dieser Typ nicht gefunden werden. Interessanterweise ist dieses Problem weder bei der Implementierung noch beim Testen mit Beispielprogrammen aufgefallen.

Schwierigkeiten beim Beweis der Vollständigkeit, die erst wie ein kleines Detail aussahen, führten nach und nach zur Konstruktion dieses Gegenbeispiels. Dies unterstreicht eindrucksvoll die Notwendigkeit einer detaillierten Beweisführung.

5.6 Homogene und inhomogene Listen

Wenn wir statt herkömmlichen Matrizen unsere Darstellung mit indizierten Typen wählen, werden manche Ausdrücke, die vorher typkorrekt waren, inkorrekt. Oft führt uns das auf Fehler im Programm, aber manchmal auch nicht. Hier folgen zwei Beispiele für Listen, die vorher homogen waren, jetzt aber inhomogen sind. In beiden Fällen kann aber die Struktur der Liste mit Indizes ausreichend beschrieben werden.

Zuerst eine Liste quadratischer Matrizen:

```
data SqMatrix a = SqMatr (Matrix a m m);

detsum :: (List (SqMatrix Int)) -> Int;
detsum [] = 0;
detsum [(SqMatr mat):xs] = (detsum xs) + (det mat);
```

und dann eine Liste von Matrizen, die multipliziert werden können:

```
data MultMatrixList a #m #n
   = MultCons (Matrix a m k) (MultMatrixList a k n)
   | MultOne (Matrix a m n);

multlist :: (MultMatrixList a m n) -> (Matrix m n);
multlist (MultOne mat) = mat;
multlist (Multcons mat ms) = mmult mat (multlist ms);
```

In beiden Fällen können wir die Information, die wir brauchen, um die gewünschte Operation auszuführen, in einen neuen Datentyp kodieren. Allerdings ist diese Kodierung mühsam und wir verlieren im zweiten Beispiel die Möglichkeit von Listenkomprehensionen.

5.7 Reihungen

Ähnlich wie bei Vektoren können wir nun auch Reihungen mit beliebiger rechter und linker Grenze definieren. Die Implementierung wird hier mit Vektoren vorgestellt, sollte aber in einer Programmiersprache wohl als Primitiv mit einer effizienten Implementierung zur Verfügung gestellt werden.

```
data Array a #l #u =
   Arr (Int' l) (Int' u) (Vector a (u - l + 1));
```

Neben diesen Reihungen sollten auch solche zur Verfügung stehen, die nach außen ihre Grenzen nicht zeigen und bei Zugriffen die Grenzen überprüfen.

```
data DynArray a = DynArr (Array a l u);
```

Für die Implementierung brauchen wir noch zwei weitere primitive Operationen auf Vektoren, die Schreibeoperation und eine Verallgemeinerung des Nullvektors, die mit einem gegebenen Wert auffüllt.

```
v_put :: (0 < m) and (m < n + 1)
      => (Vector a n) -> (Int' m) -> a -> (Vector a n);
v_put <x:xs> 1' y = <y:xs>;
v_put <x:xs> n = <x:(v_put xs (n -' 1') y)>;

v_fill :: (Int' n) -> a -> (Vector a n)
v_fill 0' x = <>;
v_fill n x = <x: (v_fill (n -' 1') x)>;
```

Nun folgen Operationen auf den Reihungen, die im wesentlichen die jeweils entsprechende Operation auf Vektoren ausführen. Zusätzlich haben wir noch die Zugriffe auf die Grenzen.

```
arr_create :: (Int' l) -> (Int' u) -> a
           -> (Array a l u);
arr_create n m x = (Arr (v_fill (m -' n) x));

arr_get :: i < h + 1, i > l - 1
        => (Array a l h) -> (Int' i) -> a;
arr_get (Arr n m v) k = v_get (k -' n) v;

arr_put :: #i < h + 1, i > l - 1
        => (Array a l h) -> (Int' i) -> a
        -> (Array a l h);
arr_put (Arr n m v) k x =
   (Arr n m (v_put (k -' n) x v));

arr_lower :: (Array a l h) -> (Int' l);
arr_lower (Arr n m v) = n;
```

```
arr_upper :: (Array a l h) -> (Int' h);
arr_upper (Arr n m v) = m;
```

Das folgende `fold` für Reihungen würde auf primitiven Reihungen genauso funktionieren, wie auf den hier mit Vektoren implementierten.

```
arr_fold :: (a -> b -> b) -> b -> (Array a l u) -> b;
arr_fold f u a = arr_fold_from f u a (arr_lower a);

arr_fold_from :: (f > l-1)
           => (a -> b -> b) -> b -> (Array a l u)
           -> (Int' f) -> b;
arr_fold_from f u from a =
      if' (less' (arr_upper a) from) then
          u
      else
          (arr_fold_from f (f (arr_get a from) u)
                       a (from +' 1'));
```

Wir können die entsprechenden Funktionen auch für Reihungen mit unbekannten Grenzen formulieren, müssen an manchen Stellen dann aber Fehler abfangen.

```
darr_create :: Int -> Int -> a -> (DynArray a);
darr_create (ExInt nst) (ExInt mst) x =
    (DynArr (arr_create nst mst x));

darr_get :: (DynArray a) -> Int -> a;
darr_get (DynArr ar) (ExInt ist) =
    if' (less' (arr_upper ar) ist) then
       error "upper bound error";
    else if' (less' ist (arr_lower ar)) then
       error "lower bound error";
    else
       (arr_get ar ist);

darr_put :: (DynArray a) -> Int -> a -> (DynArray a);
darr_put (DynArr ar) (ExInt ist) x =
    if' (less' (arr_upper ar) ist) then
```

```
         error "upper bound error";
      else if' (less' ist (arr_lower ar)) then
         error "lower bound error";
      else
         (DynArr (arr_put ar ist x));

darr_lower :: (DynArray a) -> Int;
darr_lower (DynArr ar) = (ExInt (arr_lower ar));

darr_upper :: (DynArray a) -> Int;
darr_upper (DynArr ar) = (ExInt (arr_upper ar));
```

Wir beachten, daß bei `darr_fold` *keine* Überprüfungen zur Laufzeit stattfinden, obwohl die Reihungsgrenzen dynamisch sind.

```
darr_fold :: (a -> b -> b) -> b -> (DynArray a) -> b;
darr_fold f u (DynArr ar) = arr_fold f u ar;
```

5.8 Invarianten

Wir haben vorher die Länge eines Vektors berechnet. Wir können auch einen Vektor definieren, der seine Länge explizit weiß. Dabei wollen wir nun aber sicher gehen, daß diese Länge immer mit der tatsächlichen Länge übereinstimmt. Dies können wir mit indizierten Typen so ausdrücken:

```
data SizedVector a #n = SVec (Int' n) (Vector a n);
SVcons :: a -> (SizedVector a n) -> (Sizedvector a n+1);
SVcons x (SVec n xs) = SVec (n +' 1') <x:xs>;
```

Bei der Operation `SVcons` ist jetzt sichergestellt, daß die Invariante erhalten bleibt.

5.8.1 AVL-Bäume

Auch die Gültigkeit der Balance-Bedingung bei AVL-Bäumen kann mit indizierten Typen ausgedrückt werden.

Die Information, welcher Teilbaum der höhere ist oder ob beide gleich hoch sind, ist im jeweiligen Konstruktor kodiert. `LBr` bedeutet, daß der linke Teilbaum größer ist.

```
data Avl a #h
   = Lf <= h = 0
   | LBr a (Avl a hl) (Avl a hr) <=
             hl = hr + 1, h = hl + 1
   | RBr a (Avl a hl) (Avl a hr) <=
             hr = hl + 1, h = hl + 2
   | MBr a (Avl a hl) (Avl a hr) <=
             hr = hl, h = hl + 1;
```

Die Funktionen, die AVL-Bäume bearbeiten, müssen den aufrufenden Funktionen mitteilen, ob der Ergebnisbaum die gleiche Höhe hat. Dies wird durch einen Wert vom Typ `Tag` ausgedrückt. Da das Ergebnis nur für einen bestimmten Tag stimmt, muß das Paar (AVL-Baum, Tag) noch einmal in einer eigenen Struktur gekapselt werden.

```
data Tag #t = Inc <= t = 1
            | Same <= t = 0;

data AvlT a #h = Ta (Avl a h + t) (Tag t);
```

Nun beschreiben wir die Einfügeroutine für AVL-Bäume:

```
insertAvl :: Int -> (Avl Int h) -> (AvlT Int h);
insertAvl x (Lf) = (Ta (MBr x Lf Lf) Inc);
insertAvl x (MBr y l r) =
       if (ls x y) then
          (case (insertAvl x l) of
           {
             (Ta l2 (Inc)) -> (Ta (LBr y l2 r) Inc);
             (Ta l2 (Same)) -> (Ta (MBr y l2 r) Same)
           })
       else
          (case (insertAvl x r) of
           {
             (Ta r2 (Inc)) -> (Ta (RBr y l r2) Inc);
             (Ta r2 (Same)) -> (Ta (MBr y l r2) Same)
           });

insertAvl x (LBr y l r) =
```

```
        if (ls x y) then
           (case (insertAvl x l) of
            {
              (Ta l2 (Same)) -> (Ta (LBr y l2 r) Same);
              (Ta (LBr z ll lr) (Inc))
                  -> (Ta (MBr z ll (MBr y lr r)) Same);
              (Ta (MBr z ll lr) (Inc))
                  -> (Ta (RBr z ll (LBr y lr r)) Inc);
              (Ta (RBr z ll (LBr u lrl lrr)) (Inc))
                  -> (Ta (MBr u (MBr z ll lrl)
                                (RBr y lrr r)) Same);
              (Ta (RBr z ll (MBr u lrl lrr)) (Inc))
                  -> (Ta (MBr u (MBr z ll lrl)
                                (MBr y lrr r)) Same);
              (Ta (RBr z ll (RBr u lrl lrr)) (Inc))
                  -> (Ta (MBr u (LBr z ll lrl)
                                (MBr y lrr r)) Same)
            })
        else
           (case (insertAvl x r) of
            {
              (Ta r2 (Inc)) -> (Ta (MBr y l r2) Same);
              (Ta r2 (Same)) -> (Ta (LBr y l r2) Same)
            });
```

Das Einfügen für `insertAvl x (RBr y l r)` geht analog. Das Erstellen falscher AVL-Bäume gibt jetzt einen Typfehler:

```
/* ERROR */
(RBr 2 Lf (RBr 3 Lf (MBr 1 Lf Lf)));
```

Wir können noch eine Struktur für geordnete Listen schreiben, die AVL-Bäume benutzt, aber die Komplexität vor dem Benutzer verbirgt.

```
data Ordered a = MkOrder (Avl a h);

insert :: Int -> (Ordered Int) -> (Ordered Int);
insert x (MkOrder a) =
    case (insertAvl x a) of
```

```
{
     (Ta b t) -> (MkOrder b)
};
```

5.8.2 Verkettete Listen als Vektoren

Oft werden verkettete Listen aus Effizienzgründen mit zwei Reihungen dargestellt, eine mit den Elementen und eine zweite mit der Verkettung. Eine wichtige Invariante ist, daß die Zeiger nie aus der Reihung herauszeigen. Hier folgt eine Implementierung mit indizierten Typen, die diese Invariante beweist, und daher auch zur Laufzeit auf Überprüfung der Zugriffsgrenzen verzichten kann.

`(BoundedInt n)` implementiert eine ganze Zahl zwischen 1 und n und `LinkedList` ist eine Implementierung einer verketteten Liste mit zwei Vektoren.

```
data BoundedInt #n
   = Bounded (Int' m) <= (0 < m) and (m < n + 1);
data LinkedList a m
   = Linked (Vector a m) (Vector (BoundedInt m) m);
```

Der Zugriff auf Elemente läuft rekursiv durch die Zeiger:

```
l_get :: (0 < m) and (m < n + 1)
      => (LinkedList a n) -> (Int' m) -> a;
l_get l i = l_get_rec l (Bounded 1') i
l_get_rec :: (0 < m) and (m < n + 1)
      => (LinkedList a n) -> (BoundedInt n)
      -> (Int' m) -> a;
l_get_rec (Linked v ch) (Bounded b) 1' = v_get b v;
l_get_rec  (Linked v ch) (Bounded b) n
    = l_get_rec (Linked v ch) (v_get b ch) (n -' 1');
```

5.8.3 Sortierte Bäume ganzer Zahlen

Wenn die Elemente eines geordneten Baums ganze Zahlen sind, können wir sogar die Sortiertheit als Invariante festhalten. Hier ist die Grenze zu Beweisern nahe. `(SortedIntTree l u)` beschreibt einen Baum, dessen eingetragene Werte e der Aussage $l \leq e < u$ genügen.

```
data SortedIntTree #l #u
   = SILeaf <= (l < u)
   | SIBranch (SortedIntTree l m) (Int' m)
        (SortedIntTree m u) <= (l < m) and (m < u);
```

Wir tragen einen neuen Wert ein. Die neuen Grenzen können wir beim Aufruf festlegen. Der Bereich muß nur größer werden und k beinhalten.

```
insertInt :: (k > m - 1) and (k < v) and
          (m < l + 1) and  (v > u - 1) =>
          (Int' k) -> (SortedIntTree l u)
                   -> (SortedIntTree m v);
insertInt i (SILeaf) = (SIBranch (SILeaf) i (SILeaf));
insertInt i (SIBranch s j t) =
   if' (less' i j) then
       (SIBranch (insertInt i s) j t)
   else
       (SIBranch s j (insertInt i t));
```

5.9 Vergleich mit abhängigen Typen

Wir wollen in diesem Abschnitt die Möglichkeiten von abhängigen Typen und indizierten Typen vergleichen. Dazu wollen wir zuerst einige allgemeine Unterschiede herausstellen, einen Kalkül für abhängige Typen angeben und dann versuchen, die Möglichkeiten des einen im jeweils anderen nachzubilden. Die Einbettung von indizierten Typen in abhängige Typen ist theoretisch einfach, in die andere Richtung ist eine Einbettung nicht immer möglich. Wir wollen dennoch die Möglichkeiten einer Transformation diskutieren, bleiben dabei aber eher informal.

5.9.1 Unterschiede

Was sind die wesentlichen Unterschiede zwischen indizierten und abhängigen Typen?

Abhängige Typen sind allgemeiner. Wir lassen bei indizierten Typen nur bestimmte Werte zu (in unserem Beispiel ganze Zahlen und Wahrheitswerte). Bei abhängigen Typen können Typen mit allen möglichen Werten parametrisiert werden. Auch lassen abhängige Typen beliebige Operationen auf den

Werten zu, auf den Indizes sind nur wenige Operationen erlaubt (in unserem Beispiel lineare Operationen, Gleichheit und Vergleiche).

Bei abhängigen Typen werden Typen und Aussagen fast gleich behandelt, bei indizierten Typen hingegen unterscheiden wir stark. Da wir bei indizierten Typen entscheidbare Systeme haben, müssen wir für Aussagen keine Terme bzw. Beweise annotieren. Dies ist der große Vorteil von indizierten Typen.

Außerdem unterscheiden wir bei indizierten Typen zwischen Werten und Indizes und brauchen Metaaussagen, um zu zeigen, daß `(Int' 5)` wirklich nur den Wert `5'` als Wert zuläßt. Bei abhängigen Typen gibt es diese Unterscheidung nicht.

5.9.2 Ein Kalkül

Sehen wir uns einmal die Regeln eines Kalküls mit abhängigen Typen an. Wir unterscheiden dabei nicht zwischen Typen und Typschemata und bezeichnen beide mit τ. Gedanklich stellen wir uns zwei Sorten **prop** und **type** vor, manifestieren das aber nicht im Kalkül. **prop** soll dabei für Aussagen stehen. Terme von solchen **prop**-Typen dienen nur als Beweise und haben im Endeffekt keine Bedeutung für die Berechnung.

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

$$(weak)\frac{\Gamma \vdash E : \tau}{\Gamma, x : \tau' \vdash E : \tau}$$

$$(\forall I)\frac{\Gamma, x : \tau \vdash E : \tau'}{\Gamma \vdash \lambda x.E : \forall x : \tau.\tau'}$$

$$(\forall E)\frac{\Gamma \vdash E : \forall x : \tau.\tau' \quad \Gamma \vdash F : \tau}{\Gamma \vdash EF : [F/x]\tau'}$$

$$(\forall_\alpha I)\frac{\Gamma \vdash E : \tau}{\Gamma \vdash \Lambda\alpha.E : \forall\alpha.\tau} \quad \alpha \notin \mathrm{fv}(\Gamma)$$

$$(\forall_\alpha E)\frac{\Gamma \vdash E : \forall\alpha.\tau'}{\Gamma \vdash E\tau : [\tau/\alpha]\tau'}$$

$$(let)\frac{\Gamma \vdash E : \tau \quad \Gamma, x : \tau \vdash F : \tau'}{\Gamma \vdash \textbf{let } x = E \textbf{ in } F : [E/x]\tau'}$$

$$(fix)\frac{\Gamma, x : \tau \vdash E : \tau}{\Gamma \vdash \textbf{fix } x :: \tau \textbf{ in } E : \tau}$$

$$(\textbf{data } I)\frac{\textbf{data } T\bar{\alpha}\bar{n} : \bar{\nu} = \Sigma D_i\bar{x}_i : \bar{\tau}_i, \bar{\beta}_i, \bar{m}_i : \bar{\mu}_i}{\Gamma \vdash D_i : \forall\bar{\alpha}.\forall\bar{n} : \bar{\nu}.\forall\bar{\beta}_i.\forall\bar{m}_i : \bar{\mu}_i.\forall\bar{x}_i : \bar{\tau}_i.\mathtt{T}\ \bar{\alpha}\bar{n}}$$

$$(\textbf{data } E)\frac{\begin{array}{l}\textbf{data } T\bar{\alpha}\bar{n} : \bar{\nu} = \Sigma D_i\bar{x}_i : \bar{\tau}_i, \bar{\beta}_i, \bar{m}_i : \bar{\mu}_i \\ \Gamma, \bar{x}_i : \bar{\tau}_i \vdash E_i : \tau \\ \Gamma \vdash F : \mathtt{T}\ \bar{\alpha}\bar{n}\end{array}}{\Gamma \vdash \textbf{case } F \textbf{ of } \{D_i \Rightarrow E_i\} : \tau}$$

$$(\exists I)\frac{\Gamma \vdash E : \tau \quad \Gamma \vdash F : [E/x]\tau'}{\Gamma \vdash \langle E, F\rangle : \exists x : \tau.\tau'}$$

$$(\exists E)\frac{\Gamma \vdash E : \exists x : \tau.\tau' \quad \Gamma, x : \tau, y : \tau' \vdash F : \tau''}{\Gamma \vdash (F \textbf{ where } \langle x, y\rangle = E) : \tau''} \quad x \notin \mathrm{fv}(\tau'')$$

$$(\exists_\alpha I)\frac{\Gamma \vdash F : [\tau/\alpha]\tau'}{\Gamma \vdash \langle \tau, F\rangle : \exists\alpha.\tau'} \quad \alpha \notin \mathrm{fv}(\tau')$$

$$(\exists_\alpha E)\frac{\Gamma \vdash E : \exists\alpha.\tau' \quad \Gamma, y : \tau' \vdash F : \tau''}{\Gamma \vdash (F \textbf{ where } \langle \alpha, y\rangle = E) : \tau''}$$

$$(conv)\frac{\Gamma \vdash E : \tau}{\Gamma \vdash E : \tau'} \quad \tau =_\beta \tau'$$

Die Typdeklarationen dürfen auch Abhängigkeiten wie

```
data SizedVector a = SVec n:Int (Vector a n);
```

enthalten. Für die Existenzquantoren haben wir schwache Summen verwendet.

Die Verwendung der Fixpunktregel mit Beweisobjekten scheint einer Induktion ohne Betrachtung des Basisfalls zu entsprechen.

```
(fix x::C in x):C
```

Andererseits terminiert das obige Programm nicht und für terminierende Programme gilt immer, daß ein Basisfall erreicht wurde, und ein korrektes Beweisobjekt existiert.

Die Regeln, die intuitiv zur Sorte **prop** gehören, haben wir in diesem Kalkül ausgelassen. Sie können je nach verwendetem Beweissystem sehr verschieden sein. Beispiele für solche Beweisregeln sind die Gleichheitsaxiome.

$$(refl)\frac{\Gamma \vdash E : \sigma}{\Gamma \vdash (\mathbf{refl}\ E) : (E = E)}$$

$$(trans)\frac{\Gamma \vdash p : (E = F) \quad \Gamma \vdash q : (F = G)}{\Gamma \vdash (\mathbf{trans} pq) : (E = G)}$$

$$(symm)\frac{\Gamma \vdash p : (E = F)}{\Gamma \vdash (\mathbf{symm}\ p) : (F = E)}$$

Genauso müßten wir je nach Beweissystem nun zum Beispiel auch die Axiome für $<$ angeben.

5.9.3 Einbettung von indizierten Typen

Die Regel $(\exists I_l)$ des Kalküls für indizierte Typen entspricht hier einer abgeleiteten Regel.

$$\frac{\dfrac{\Gamma, p : C \vdash E : \sigma}{\Gamma, q : \exists\bar{\alpha}.\exists\bar{n}.C, p : C \vdash E : \sigma} \quad \Gamma, q : \exists\bar{\alpha}.\exists\bar{n}.C \vdash q : \exists\bar{\alpha}.\exists\bar{n}.C}{\Gamma, q : \exists\bar{\alpha}.\exists\bar{n}.C \vdash (E\ \mathbf{where}\ \langle\alpha, p\rangle = q) : \sigma}$$

Das selbe gilt für die $(=)$-Regel:

$$(=)\frac{\Delta, \Gamma \vdash E : \sigma \quad \Delta \vdash p : \Gamma = \Gamma', q : \sigma = \sigma'}{\Delta, \Gamma' \vdash (\mathbf{cut}\ pqE) : \sigma'}$$

Wir hatten in unserem System für indizierte Typen die Regel

$$(\forall \Rightarrow I)\frac{C \wedge C', \Gamma \vdash E : \tau}{C \wedge \exists\bar{\alpha}.\exists\bar{n}.C', \Gamma \vdash E : \forall\bar{\alpha}.\forall\bar{n}.C' \Rightarrow \tau} \quad \bar{n}, \bar{\alpha} \notin \mathrm{fv}(C, \Gamma)$$

gewählt, weil wir dann auch bei strikter Semantik vor Typfehlern zur Laufzeit sicher sind. Wenn wir über korrekte Programme reden, und nur für korrekte Programme ist die Ausdrucksmächtigkeit wichtig, ist der Unterschied nicht relevant, und wir können die Regel ohne $\exists\bar{\alpha}.\exists\bar{n}.C'$ als äquivalent sehen [OSW98].

Wir können nun indizierte Typen leicht in diesen Kalkül einbetten, müssen dazu aber die Beweise für unsere Aussagen explizit machen. Dies kann auch ein Programm übernehmen, da wir nur entscheidbare Systeme betrachten. Schließlich müssen wir noch explizite Typquantoren einführen, aber da wir eigentlich ganze Derivationen einbetten, ist auch dies einfach.

In die andere Richtung funktioniert so eine Einbettung im allgemeinen nicht. Wir wollen hier trotzdem eine Vorgehensweise angeben, von einem Programm im expliziten Kalkül zu einer möglichst guten Nachbildung mit indizierten Typen zu kommen. Wir werden auftretende Probleme jeweils im entsprechenden Abschnitt besprechen. Die Vorgehensweise ist kein Algorithmus und oft indeterministisch.

5.9.4 Ersetzen von Existenzquantoren

Existenzquantoren sind in diesem Kalkül nur eine bequeme Schreibweise und können durch algebraische Typen ersetzt werden. Wir ersetzen jeweils die linke durch die rechte Seite.

$\exists x : \tau.\tau'$	**data** `Exists` $\bar{\alpha} = Dx : \tau\tau'$
$\exists\beta.\tau'$	**data** `Exists` $\bar{\alpha} = D\beta\tau'$
$\langle E, F\rangle$	(DEF)
$\langle \tau, F\rangle$	$(D\tau F)$
$(F$ **where** $\langle x, y\rangle = E)$	**case** E **of** $\{(Dxy) \Rightarrow F\}$
$(F$ **where** $\langle \alpha, y\rangle = E)$	**case** E **of** $\{(D\alpha y) \Rightarrow F\}$

Es muß für jeden vorkommenden existentiellen Typ je ein neuer Typkonstruktor eingeführt werden.

5.9.5 Eigenschaften als Datentyp

Wir kodieren nun alle Eigenschaften (alle Typen der Sorte **prop**) als algebraischen Datentyp, zum Beispiel

```
data Equal #n #m = EqPrf <= (n = m);
```

Ein Term dieses Typs kann nur dann konstruiert werden, wenn wir wissen, daß $n = m$ gilt. Nach einem case mit dem Beweis als Argument können wir die Bedingung wieder verwenden.

Im allgemeinen führen wir einen neuen Konstruktor **data** $C\bar{\alpha}\bar{n} = C \Leftarrow \tau$ für die Aussage τ ein. Zur Laufzeit besteht der Term nur aus einem einzelnen Konstruktor, und auch der kann später in vielen Fällen eliminiert werden.

Wir übergeben diesen Konstruktor immer, wenn wir einen Beweis dieser Aussage brauchen. Wenn ein Datentyp dieser Art übergeben wird, schreiben wir als äußerstes Konstrukt ein case, damit die Aussage möglichst überall verwendet werden kann.

Ein Problem entsteht hier, wenn wir nicht-lineare Bedingungen haben. Betrachten wir folgende Aussage für den ganzzahligen Anteil einer Wurzel,

```
data Sqrt n m = SqrtPrf <= (m*m < n+1) and (n < (m+1)*(m+1))
                           and (0 < m+1) and (0 < n+1)
```

die wir zum Beispiel für das Ergebnis der Wurzelfunktion verwenden würden. Das können wir in Presburger Arithmetik nicht formulieren, also schreiben wir

```
data Sqrt n m = SqrtPrf <= (0 < m+1) and (m < n+1);
```

Da es sich bei den Bedingungen um Invarianten des Datentyps handelt, ist als Approximation nur eine Abschwächung möglich.

Die Beweisanteile von Termen können weggelassen werden. (**refl**E) wird zum Beispiel überall einfach durch den Konstruktor `(EqPrf)` ersetzt. Der Beweis der Transitivität der Gleichheit schreibt sich dann so

```
trans :: (Equal a b) -> (Equal b c) -> (Equal a c);
trans (EqPrf) (EqPrf) = (EqPrf);
```

Dieses Weglassen von Beweisen ist der wesentliche Vorteil von indizierten Typen.

5.9.6 Elimination von Abhängigkeiten

Wir könnten alle Abhängigkeiten von Werten aus den Typen entfernen, indem wir bei allen Typen, die mit Werten parametrisiert sind diese Werte streichen. Damit bekämen wir ein typkorrektes Programm. Dies kann allerdings nicht alleiniges Ziel sein, da alle Aussagen verloren gehen und gar nicht oder nur dynamisch überprüft werden.

Außerdem müßten wir dann primitive Typen mit Indizes wie zum Beispiel Reihungen durch Typen ohne Indizes, die manche Überprüfungen erst dynamisch vornehmen, ersetzen.

Wir versuchen also möglichst wenig Abhängigkeiten zu entfernen und möglichst viele Werte als Indizes zu kodieren. Wenn aber ein Typ von Werten abhängt, die nicht als Index beschreibbar sind (von `Int` und `Bool` verschieden), muß diese Abhängigkeit entfernt werden.

Die Entfernung eines Parameters n aus einem Typ τ entspricht in etwa einer existentiellen Quantifikation $\exists n.\tau$.

Wenn wir einen Werteparameter als Index beschreiben wollen, ersetzen wir abhängige Typen wie folgt durch indizierte Typen (Wir verwenden hier `Int`, könnten aber genauso gut `Bool` verwenden):

$$\begin{array}{rr} \forall x : \texttt{Int}.\tau' & \forall x.(\texttt{Int}'\ x) \to \tau' \\ \textbf{data}\ \ldots = D\ (x : \texttt{Int})\ \tau' & \textbf{data}\ \ldots = D\ (\texttt{Int}'\ x)\tau' \end{array}$$

Wenn wir diese Ersetzung durchführen, entstehen zwei Arten von Typfehlern. An einer Stelle, an der wir bisher einen Ausdruck `E` des Typs `Int` verwendet haben, hat `E` nun den Typ `(Int' n)`. Hier behebt eine Ersetzung von `E` durch `(ExInt E)` sofort das Problem. Schwieriger ist die andere Richtung, wenn nämlich ein Ausdruck `E` des Typs `Int` steht, wo ein Ausdruck des Typs `(Int' n)` erwartet wird. Wir ersetzen dann zuerst einmal `E` durch

```
case E of { (ExInt x) => x; }
```

Auch nach dieser Ersetzung hat der Ausdruck immer noch einen Typfehler, da der Index von `x` nicht bekannt ist. Das grundlegende Problem ist hier, daß Informationen, die bei abhängigen Typen in Werten weitergegeben werden, bei indizierten Typen im Typ kodiert werden müssen. Diese Informationskodierung im Typ und ein Ausklammern von case-Ausdrücken, wie es im

folgenden besprochen wird, führt oft zum Erfolg. Wenn der Typfehler trotzdem nicht beseitigt werden kann, muß der Werteparameter doch ganz aus dem Typ gestrichen werden.

5.9.7 Information in den Typ verlagern

Wir haben eben gesehen, daß der Typinferenz nach der Elimination der Abhängigkeiten manchmal Informationen fehlen, um die Typkorrektheit abzuleiten. Manchmal läßt sich dieses Problem beseitigen, indem wir die case-Konstruktion möglichst weit außen verwenden oder zusätzliche Information im Typ kodieren. Betrachten wir folgendes Beispiel:

```
f :: All n:Int. (Vector Int n);
g :: Int -> Int -> Int;
g n k = sprod (f (twpl n k)) (f (twpl n k));
twpl :: Int -> Int -> Int;
twpl n m = n + m + n;
```

Wenn wir die obigen Transformationen ausführen, haben wir folgendes Programm:

```
f :: (Int' n) -> (Vector Int n);
g :: Int -> Int -> Int;
g n k = sprod (f case (twpl n k) of
                { (ExInt m) -> m; })
              (f case (twpl n k) of
                { (ExInt j) -> j; })
```

Dieses Programm hat Typfehler, weil der Ergebnistyp der case-Konstrukte je eine existentielle Variable enthält. Wir ziehen nun die case-Konstrukte soweit wie möglich nach außen.

```
f :: All n.(Int' n) -> (Vector Int n);
g :: Int -> Int -> Int;
g n k = case (twpl n k) of
   {
      (ExInt m) -> case (twpl n k) of
         {
            (ExInt j) -> sprod (f m) (f j);
         }
   }
```

Hier ist Vorsicht geboten, die Äquivalenz von

```
E (case F of { (ExInt m) => G })
```

und

```
(case F of { (ExInt m) => (EG) })
```

ist bei fauler Auswertung nicht garantiert!

In diesem speziellen Fall könnten wir ausnützen, daß `(twpl n k)` ein gemeinsamer Teilausdruck ist, wollen das aber vorerst nicht tun. Das Programm hat nach wie vor einen Typfehler, aber das Problem ist jetzt ein anderes: Wir können nicht sehen, daß `(f m)` und `(f j)` die selbe Länge haben. Hier muß `twpl` geändert werden und die zusätzliche Information transportieren.

```
twpl :: (Int' n) -> (Int' m) -> (Int' k);
twpl n m = case (ExInt n) + (ExInt m) + (ExInt n) of
               { (ExInt k) => k }
```

Dies können wir durch

```
twpl n m = case (ExInt (n +' m +' n)) { (ExInt k) => k }
```

ersetzen, und mit einer weiteren Umformung erhalten wir schließlich:

```
twpl :: (Int' n) -> (Int' m) -> (Int' n+m+n);
twpl n m = (n +' m +' n);
```

Insgesamt gibt das

```
f :: (Int' n) -> (Vector Int n);
g :: Int -> Int -> Int;
g (ExInt n) (ExInt k) = sprod (f (twpl n k)) (f (twpl n k));
twpl :: (Int' n) -> (Int' m) -> (Int' n+m+n);
twpl n m = (n +' m +' n);
```

und das ist wohlgetypt. Wenn `twpl` nicht linear wäre, könnte in diesem Fall, wie bereits oben erwähnt, ausgenutzt werden, daß `(twpl n k)` ein gemeinsamer Teilausdruck ist:

```
f :: (Int' n) -> (Vector Int n);
g :: Int -> Int -> Int;
g n k = case (twpl n k) of
   {
      (ExInt m) -> sprod (f m) (f m);
   }
```

ist ebenfalls wohlgetypt.

5.9.8 Quantoren nach außen ziehen

Wir haben vorher alle Existenzquantoren eliminiert, bzw. durch algebraische Datentypen ersetzt. Jetzt müssen wir noch dafür sorgen, daß die Allquantoren in den Typschemata immer ganz außen stehen. Falls das noch nicht der Fall ist, können wir wegen der Gleichung

$$P \to (\forall \alpha.Q) \equiv \forall \alpha.P \to Q$$

folgendermaßen vorgehen: Wenn $E : \tau \to \forall \alpha.\tau'$ dann ist $\Lambda\alpha.\lambda x.(Ex\alpha) : \forall \alpha.(\tau \to \tau')$. Ein Beispiel wäre die Funktion `concat` mit dem Typ

$$\begin{aligned}&\forall n : \texttt{Int}.\forall m : \texttt{Int}.(\texttt{Vector Int}\ n)\\ &\quad \to (\texttt{Vector Int}\ m) \to (\texttt{Vector Int}\ (n+m))\end{aligned}$$

Wir können $\forall m$ natürlich ganz herausziehen:

$$\begin{aligned}&\forall n, m.(\texttt{Int}'\ n) \to (\texttt{Int}'\ m) \to (\texttt{Vector Int}\ n)\\ &\quad \to (\texttt{Vector Int}\ m) \to (\texttt{Vector Int}\ (n+m))\end{aligned}$$

Wir haben das im vorigen Abschnitt bereits benutzt, ohne es zu erwähnen. Genau so können wir auch $(\exists \alpha.\tau) \to \tau'$ als $\forall \alpha.(\tau \to \tau')$ behandeln.

Wenn wir es geschafft haben, alle Allquantoren ganz nach außen zu bringen, können wir jetzt die entsprechenden $\Lambda\alpha$ einfach weglassen.

Bei der Verwendung solcher logischer Hilfsmittel müssen wir allerdings vorsichtig sein, da wir nur intuitionistische Aussagen verwenden dürfen, und in intuitionistischer Logik wäre beispielsweise

$$(\forall \alpha.Q) \to P \not\equiv \exists \alpha.(Q \to P)$$

nicht garantiert.

Diese Problematik taucht hier auf, weil wir uns mit einer Obermenge von System F auseinandersetzen [OL96] [Jon97]. Wir wollen uns damit hier nicht weiter befassen, sehen aber eine Erweiterung in diese Richtung als interessante Weiterentwicklung für indizierte Typen.

5.9.9 Überflüssige Datentypen eliminieren

Datentypen wie `(Equal m n)`, die reine Bedingungen sind, sollten als Argumente weggelassen werden und durch ein case an der Aufrufstelle ersetzt werden. Die entsprechenden case auf den Argumenten können dann weggelassen werden. In

```
vappend :: (Equal m (n+k)) -> (Vector a n) -> (Vector a k)
        -> (Vector a m);
vappend (EqPrf) <> xs = xs;
vappend (EqPrf) <x:xs> ys = <x:(vappend (EqPrf) xs ys>;
```

können wir die Bedingung zu `m = n+k` vereinfachen und alle Beweisobjekte streichen. Dies impliziert eine Streichung bei allen Aufrufen von `vappend`.

```
vappend p xs ys
```

muß durch

```
case p of
{
   (EqPrf) -> (vappend xs ys)
}
```

ersetzt werden, dafür kann aber im Rumpf von `vappend`

```
case q of { (EqPrf) => E }
```

durch `E` ersetzt werden. Im rekursiven Aufruf bleibt nach diesen beiden Umformungen gerade die Streichung von `(EqPrf)`:

```
vappend :: m=n+k => (Vector a n) -> (Vector a k)
        -> (Vector a m);
vappend <> xs = xs;
vappend <x:xs> ys = <x:(vappend xs ys>;
```

Auch an den anderen Verwendungsstellen von `vappend` wird das case in der Regel durch eine solche Umformung wegfallen.

5.10 Verwandte Arbeiten

5.10.1 Pfenning und Xi

Xi und Pfenning [XP98], [Xi98] haben unabhängig einen sehr ähnlichen Ansatz gewählt. Die Übereinstimmung geht so weit, daß auch sie zum Beispiel `(Int' n)` Typen einführen, und ihre Anwendungen überdecken sich teilweise mit unseren.

Eines ihrer wichtigen Ziele ist die Wiederverwendung existierender ML-Programme in ihrer Erweiterung. Ein Vorteil ihres Systems ist es, daß `Int` nicht mit Hilfe eines neuen Konstruktors aus `(Int' n)` gewonnen werden muß, `Int` ist ein Synonym für (∃`n`.`Int n`). Existierende ML-Programme sind also auch in der Erweiterung typkorrekt.

Als Folge davon ist ihre Typinferenz komplizierter. Ein wesentlicher Vorteil unseres Systems hingegen ist, daß wir Vollständigkeit gegenüber einem einfachen logischen System zeigen können. Dies läßt den Benutzer leichter nachvollziehen, was typkorrekt ist. Xi und Pfenning können nur zeigen, daß ihr System eine konservative Erweiterung von ML ist.

5.10.2 Sized Types

Wir wollen nun indizierte Typen auch noch mit *sized types* von Hughes, Pareto und Sabry [HPS96] vergleichen. Die Motivation von sized types war es, verschiedene Probleme im Bereich reaktiver Systeme wie Produktivität und Terminierung statisch zu lösen.

Bei sized types können Indizes nicht vom Programmierer frei gewählt werden, sondern beschreiben immer die Schachtelungstiefe der Konstruktoren eines Typs. Das entstehende System ist daher nicht so flexibel wie indizierte Typen.

Die Induktion zur Korrektheit der Typisierung geht bei sized types über den Index, der immer eine natürliche Zahl ist. Wir verwenden ganze Zahlen und die Induktion zur Korrektheit geht über die Tiefe der Rekursion. Als Konsequenz aus dieser Tatsache können wir keine Aussagen über die Terminierung machen und die Typkorrektheit eines Ergebnisses ist bei indizierten Typen nur im Fall der Terminierung gegeben. Dabei können natürlich auch keine Typfehler auftreten.

Ein weiteres Problem von sized types ist, daß die Funktion `reverse` mit akkumulierendem Parameter nicht typisiert werden kann.

```
vreverse :: (Vector a n) -> (Vector a n);
vreverse xs = vrev xs <>;
vrev :: (Vector a n) -> (Vector a m) -> (Vector a (n + m))
vrev <> ys = ys;
vrev <x:xs> ys = vrev xs <x:ys>
```

Das Problem ist, daß bei sized types versucht wird, mit einer Induktion über n und m die Terminierung zu zeigen, m aber in den rekursiven Aufrufen von

`vrev` wächst. Bei indizierten Typen geht der Beweis über die Rekursionstiefe und das führt in diesem Fall zum Erfolg.

5.10.3 Cayenne

Cayenne ist eine funktionale Sprache mit expliziten abhängigen Typen. Die Sprache ist mächtiger als indizierte Typen, und eher dem Beispielkalkül für abhängige Typen vergleichbar als unserer Programmiersprache. Die Typinferenz ist aber unentscheidbar.

Das Ziel von Cayenne ist eher, zusätzliche Konstrukte wie `printf` zu typisieren, während das Ziel von indizierten Typen ist, mehr Fehler bereits zur Übersetzungszeit zu finden.

5.10.4 Shape Types

S hape types wurden von Jay für die Programmiersprache FISh entwickelt [JS98], [Jay98]. Das Ziel ist, anders als bei uns, im wesentlichen Effizienzgewinn, und nicht das Finden von Fehlern.

Ähnlich wie bei sized types ist der Index durch den Typ festgelegt, hier die Form (shape) des Typs. Außerdem wird der Benutzer in der Verwendung höherwertiger Funktionen eingeschränkt, Funktionen dürfen zum Beispiel nicht in Reihungen vorkommen.

Kapitel 6

Zusammenfassung und Ausblick

Am Ende dieser Arbeit wollen wir noch einmal das Erreichte zusammenfassen und einen Ausblick auf mögliche Weiterentwicklungen geben. Außerdem gehen wir kurz auf die Implementierung ein.

6.1 Zusammenfassung

Wir haben in dieser Arbeit ein Typsystem für indizierte Typen vorgestellt. Mit diesem Typsystem ist es möglich, viele Probleme von Typen mit Werteparametern zu beschreiben. Wir haben Beispiele aus verschiedenen Bereichen gesehen, vorrangig aus der linearen Algebra, aus Anwendungen mit Reihungen und bei Invarianten von Datenstrukturen.

Indizierte Typen stehen zwischen den Typsystemen von Beweisern mit abhängigen Typen und einem klassischen Hindley-Milner-Typsystem wie in ML. Wir haben die entscheidbare Typinferenz von ML, zusätzlich aber die Möglichkeit von Werteparametern. Indizierte Typen haben nicht die Allgemeinheit der abhängigen Typen eines Beweisers, sind aber für viele praktische Probleme mit Wertparametern geeignet. Außerdem läßt sich in dem vorgestellten System leichter programmieren als mit einem Beweiser, da viel mehr inferiert wird.

Auf der technischen Seite haben wir die auftretenden Probleme gelöst. Wir haben konzeptuell in Inferenz und Unifikation aufgeteilt. Viele Typinferenzalgorithmen verwenden die Vereinigung von Constraintmengen, die einem logischen *und* der enthaltenen Constraints entspricht. Wir müssen in der case-Regel sozusagen Constraints aus einer Constraintmenge herausnehmen.

Dafür haben wir die Implikation $P \to C$ auf Constraints als geeignete Operation gefunden: $P \to C$ ist der schwächste Constraint, der zusammen mit $P\ C$ impliziert.

Weiter haben wir die bereits in der Einleitung erwähnte Schwierigkeit, daß ein Argumentmuster immer den allgemeinen Argumenttyp haben muß durch eine Unterscheidung zwischen Gleichheit und struktureller Gleichheit gelöst.

Die Unifikation ist durch die zusätzlichen Implikationen, Indexbedingungen und Quantoren erheblich komplizierter geworden. Dennoch konnten wir einen Lösungsalgorithmus angeben.

Wir haben Korrektheit und Vollständigkeit von Typinferenz und Unifikation bewiesen, und ebenso die operationelle Korrektheit des Systems. Der Beweis für die Unifikation ist völlig neu, die Beweise für Korrektheit und Vollständigkeit der Typinferenz stützen sich natürlich auf entsprechende Beweise für einfachere Systeme, aber zumindest die Erweiterung um die case-Regel ist nicht trivial.

6.2 Implementierung

Wir haben einen Prototyp für eine funktionale Programmiersprache mit indizierten Typen implementiert. Die Sprache orientiert sich, wie auch die Beispiele in dieser Arbeit, stark an der Syntax von Haskell [PHA+97]. Der Parser, die Musterüberprüfung und die Erzeugung der Typconstraints wurde mit Hilfe der Compilerbau Tool Box Cocktail [GE90] in C implementiert. Die Unifikation selbst wurde in Pizza [OW97] implementiert. Sie vereinfacht ein Unifikationsproblem zu einem Termconstraint in Presburger Arithmetik, der mit Hilfe von Omega [Pug92] auf Erfüllbarkeit überprüft werden kann.

6.3 Ausblick

Wohin könnte die Entwicklung weitergehen? Welche Fragen bleiben? Wir wollen das hier in praktische und theoretische Fragen aufteilen.

Praktische Fragen

Wir haben eine Prototypimplementierung des Systems. Um aber die Möglichkeiten des Typsystems an größeren Studien zu untersuchen, wäre eine aus-

gereiftere und effizientere Implementierung notwendig.

Erst dann läßt sich zum Beispiel feststellen, in wie weit durch die fehlenden Grenzüberprüfungen bei Reihungen ein Effizienzgewinn erzielt werden kann. Dazu wäre auch eine effiziente Implementierung von Reihungsmonaden erforderlich.

Des weiteren wäre eine Studie interessant, die analysiert, in wie weit das Arbeiten mit indizierten Typen das Programmieren selbst effizienter und die entstehenden Programme robuster macht. Wieviel helfen Bibliotheken, die mit indizierten Typen geschrieben sind?

Theoretische Fragen

Die wichtigste Frage scheint hier zu sein, wie gut sich indizierte Typen mit Überladung kombinieren lassen. Auf den ersten Blick scheint das leicht integrierbar, und die folgende Definition scheint natürlich

```
sprod :: Num a => (Vector a n) -> (Vector a n) -> a
```

Aber wir können Matrizen nicht ohne weiteres als Instanz von `Num` deklarieren, weil die Multiplikation beliebiger Matrizen ja nicht erlaubt ist. Wir würden gerne eine Typklasse `Mult` als

```
class Mult a where
   (*) :: a -> a -> a
```

definieren und als Instanz

```
instance Num a => Mult (Matrix a n n)
```

angeben. Die Inferenz mit Typklassen, wenn die Klassenzugehörigkeit von den Indizes abhängt, muß untersucht werden. Ist es günstiger, Typklassen [PHA+97], [Jon94] oder System O [OWW95] zu integrieren? Können wir vielleicht sogar Konstruktorklassen [Jon95] einführen?

Die zweite Frage ist, ob wir explizite Quantoren einführen können. Insbesondere Existenzquantoren in Funktionsresultaten müssen in immer wieder neuen Datenstrukturen beschrieben werden. Dies scheint möglich, aber die Details müssen ausgearbeitet werden [OL96].

Der dritte Punkt von vorrangigem Interesse ist die Mustervollständigkeit. Bei indizierten Typen ist es möglich, daß Muster für Argumente angegeben sind, die unter den Bedingungen, die für die Funktion inferiert wurden, niemals aufgerufen werden können. Ein Beispiel wäre

```
vihead :: (n > 0) => (Vector Int n) -> Int;
vihead (Vcons x xs) = x;
vihead (Vnil) = 0;
```

Die letzte Zeile ist offensichtlich irrelevant. Sollten wir hier nicht einen Fehler melden? Ist es möglich, die Typinferenz so zu erweitern?

Auf der anderen Seite gibt es ganz normale Musteranpassungsfehler wie der Aufruf `(vtail Vnil)` in

```
vtail :: (Vector a n) -> (Vector a (n - 1))
vtail (Vcons x xs) = xs;
```

Können wir Mustervollständigkeit verlangen? Das heißt, dürften nur die Muster fehlen, die durch die angegebene Bedingung ausgeschlossen sind? In dem gerade geschilderten `vtail` müßte dann entweder ein `Vnil` Zweig hinzukommen oder eine schärfere Bedingung formuliert werden. Ersteres ist nicht möglich, weil wir keinen Vektor der Länge -1 konstruieren können. In diesem Fall müßten wir also die Bedingung verschärfen.

```
vtail :: (n > 0) => (Vector a n) -> (Vector a (n - 1))
vtail (Vcons x xs) = xs;
```

Wäre das wirklich eine Bereicherung oder würde das in praktischen Fällen behindern? Wäre vielleicht eine Warnung geeignet? Zuerst muß allerdings die Inferierbarkeit untersucht werden.

Als letztes wollen wir hier noch kurz auf eine mögliche Erweiterung auf imperative Sprachen eingehen. Das Hauptproblem ist hierbei das folgende. Bei einer funktionalen Programmiersprache gibt es für jede Variable genau einen Typ und genau einen Wert. Bei einer imperativen Sprache dagegen wird einer Variable zwar auch nur ein Typ gegeben, aber ihr werden manchmal mehrere Werte zugewiesen. Dies führt dann zu Problemen, wenn Typen Werte sehr differenziert beschreiben, und bestimmte Zuweisungen aus Typgründen nicht möglich sind. Ein Beispiel wäre die Zuweisung

```
i: Int' n;
i := i + 1;
```

Dies kann vielleicht dadurch erlaubt werden, daß sich Indizes von Typen ändern dürfen und vielleicht nur in einer Static-Single-Assignment Form fest sind. Das zweite Problem ist, daß bei indizierten Typen für jede Rekursion

ein Typ deklariert werden muß. Weil wir viele Funktionen höherer Ordnung verwenden, kommt das im funktionalen nicht so oft vor. Im Imperativen entspricht dies einer Angabe der Schleifeninvariante für Indizes. Aber diese kann ja vielleicht oft, zum Beispiel in einer for-Schleife, implizit erfolgen.

Anhang A

Beweise

In diesem Anhang sind die Beweise der Arbeit zusammengefaßt. Bei einfachen strukturellen Induktionen oder semantischen Überprüfungen ist oft nur die Beweismethode angegeben.

A.1 Eigenschaften von $\vdash^A$ und $\bullet$

Eigenschaften von $\vdash^A$

Proposition 7 (Eigenschaften von $\vdash^A$). *Folgende Regeln sind semantisch korrekt. Wenn also die obere Aussage semantisch korrekt ist, ist das auch die untere.*

$$(taut)\frac{}{C \vdash^A C}$$

$$(\wedge I)\frac{C \vdash^A A \quad C \vdash^A B}{C \vdash^A A \wedge B} \quad (\wedge E_r)\frac{C \vdash^A A \wedge B}{C \vdash^A A} \quad (\wedge E_l)\frac{C \vdash^A A \wedge B}{C \vdash^A B}$$

$$(\to E)\frac{C \vdash^A P \quad C \vdash^A P \to B}{C \vdash^A B} \quad (\to I)\frac{C \wedge P \vdash^A B}{C \vdash^A P \to B}$$

$$(\forall I)\frac{C \vdash^A A}{C \vdash^A \forall\alpha.A}\alpha \notin \mathrm{fv}(C) \quad (\forall E)\frac{C \vdash^A \forall\alpha.A}{C \vdash^A [a/\alpha]A}$$

$$(\exists I)\frac{C \vdash^A [a/\alpha]A}{C \vdash^A \exists\alpha.A} \quad (\exists E)\frac{C \vdash^A \exists\alpha.A \quad C \wedge A \vdash^A B}{C \vdash^A B}\alpha \notin \mathrm{fv}(C,B)$$

$$(=_\tau I_1)\frac{C \vdash^A \tau_1 = \tau_1', \tau_2 = \tau_2'}{C \vdash^A \tau_1 \to \tau_2 = \tau_1' \to \tau_2'} \quad (=_\tau E_1)\frac{C \vdash^A \tau_1 \to \tau_2 = \tau_1' \to \tau_2'}{C \vdash^A \tau_1 = \tau_1', \tau_2 = \tau_2'}$$

$$(=_\tau I_2)\frac{C \vdash^A \bar{\tau} = \bar{\tau}', \bar{t} = \bar{t}'}{C \vdash^A T\bar{\tau}\bar{t} = \bar{\tau}'\bar{t}'} \quad (=_\tau E_2)\frac{C \vdash^A T\bar{\tau}\bar{t} = \bar{\tau}'\bar{t}'}{C \vdash^A \bar{\tau} = \bar{\tau}', \bar{t} = \bar{t}'}$$

$$(=_t refl)\frac{}{C \vdash^A t = t} \quad (=_\tau refl)\frac{}{C \vdash^A \tau = \tau}$$

$$(=_t trans)\frac{C \vdash^A t = t', t' = t''}{C \vdash^A t = t''} \quad (=_\tau trans)\frac{C \vdash^A \tau = \tau', \tau' = \tau''}{C \vdash^A \tau = \tau''}$$

$$(=_t sym)\frac{C \vdash^A t = t'}{C \vdash^A t' = t} \quad (=_\tau sym)\frac{C \vdash^A \tau = \tau'}{C \vdash^A \tau' = \tau}$$

Beweis:
Diese Eigenschaften sind leicht mit der Semantik zu zeigen. Wir führen dies an zwei ausgewählten Beispielen vor. Die anderen Eigenschaften können nach dem selben Schema bewiesen werden.

$$\frac{C \vdash^A A \quad C \vdash^A B}{C \vdash^A A \wedge B}$$

Wir nehmen an, es gilt $\pi_i([\![C]\!]_{A,\rho})$. Also gilt nach der ersten Prämisse $\pi_i([\![A]\!]_{A,\rho})$ und nach der zweiten $\pi_i([\![B]\!]_{A,\rho})$. Somit gilt nach Definition $\pi_i([\![A \wedge B]\!]_{A,\rho})$, was zu zeigen war.

$$\frac{C \vdash^A \exists\alpha.A \quad C \wedge A \vdash^A B}{C \vdash^A B}\alpha \notin \mathrm{fv}(A,B)$$

Wieder nehmen wir $\pi_i([\![C]\!]_{A,\rho})$ an. Dann haben wir mit der ersten Prämisse $\pi_i([\![\exists\alpha.A]\!]_{A,\rho})$, also $\exists a.\pi_i([\![A]\!]_{A,\rho[a/\alpha]})$. Wir wählen ein solches a und schließen (a kommt in C nicht vor) $\pi_i([\![C \wedge A]\!]_{A,\rho[a/\alpha]})$. Nun bekommen wir mit der zweiten Prämisse $\pi_i([\![B]\!]_{A,\rho[a/\alpha]})$, was aber das gleiche ist wie $\pi_i([\![B]\!]_{A,\rho})$ (wieder kommt a nicht in B vor). □

Hilfssatz 1 (Abgeleitete Eigenschaften von $\vdash^A$). *Auch die folgenden sind Regeln sind semantisch gültig:*

$$(trans)\frac{C \vdash^A A \quad A \vdash^A B}{C \vdash^A B}$$

$$(cut)\frac{C \wedge A \vdash^A B \quad C \vdash^A A}{C \vdash^A B}$$

$$(\exists lr)\frac{C \vdash^A C'}{\exists\alpha.C \vdash^A \exists\alpha.C'}$$

$$(\forall lr)\frac{C \vdash^A C'}{\forall\alpha.C \vdash^A \forall\alpha.C'}$$

$$(P \rightarrow)\frac{A \vdash^A B}{P \rightarrow A \vdash^A P \rightarrow B}$$

Beweis:
Wir können dies wieder mit einfachen semantischen Überprüfungen beweisen, die hier wieder nicht ausgeführt werden. □

Wir vermuten, daß wir diese Eigenschaften auch aus den Eigenschaften aus Proposition 7 syntaktisch herleiten könnten. Dies würde vermutlich die ein oder andere strukturelle Induktion über Constraints erfordern.

Hilfssatz 2 (Gleiches durch Gleiches ersetzen). *Wir können in einer Aussage Gleiches durch Gleiches ersetzen.*

$$\frac{A \equiv^A B}{C[A] \vdash^A C[B]}$$

Die Formel C darf dabei $\bullet$ nicht enthalten.

Beweis:
Der Beweis kann leicht erbracht werden, indem $\pi_i([\![C[A]]\!]_{A,\rho}) = \pi_i([\![C[B]]\!]_{A,\rho})$ durch Induktion über die Struktur von C gezeigt wird. □

Wir vermuten, daß die Aussage auch für Formeln mit $\bullet$ gilt, der Beweis scheint aber schwierig.

Hilfssatz 3 (Einige Äquivalenzen).

$$\begin{aligned}
\exists\alpha.(A \wedge B) &\equiv^A A \wedge \exists\alpha.B, \qquad \alpha \notin \mathrm{fv}(A)\\
\forall\alpha.(A \wedge B) &\equiv^A \forall\alpha.A \wedge \forall\alpha.B\\
\forall\alpha.A &\equiv^A A, \qquad \alpha \notin \mathrm{fv}(A)\\
P' \to (P \to C) &\equiv^A (P \wedge P') \to C\\
P \to (C \wedge C') &\equiv^A (P \to C) \wedge (P \to C')
\end{aligned}$$

Beweis:
Auch diese Äquivalenzen sind trivial, wenn wir jeweils die Semantik der linken und rechten Seite betrachten. □

Hilfssatz 4 (Substitutionen). *Wenn ψ und $[\bar\tau/\bar\alpha]$ idempotente Substitutionen sind gilt:*

$$[\![[\bar\tau/\bar\alpha]\tau']\!]_\rho = [\![\tau']\!]_{\rho[[\![\bar\tau]\!]_\rho/\bar\alpha]}$$

$$[\![[\bar\tau/\bar\alpha]C]\!]_\rho = [\![C]\!]_{\rho[[\![\bar\tau]\!]_\rho/\bar\alpha]}$$

$$\frac{\psi \wedge \psi C \vdash \psi C'}{\psi C \vdash \psi C'}$$

$$\psi \wedge C \equiv^A \psi \wedge \psi C$$

$$\frac{C \vdash C'}{\psi C \vdash \psi C'}$$

$$\psi \vdash^A \psi\tau = \tau$$

Beweis:
Die ersten beiden Behauptungen sind leicht mit struktureller Induktion über τ beziehungsweise C zu zeigen. Die dritte Behauptung gilt wegen

$$\frac{\psi \wedge \psi C \vdash \psi C'}{\exists\,\mathrm{dom}(\psi).\psi \wedge \psi C \vdash \psi C'},$$

und der Rest ist wieder leicht durch Betrachten der Semantik zu sehen. □

Eigenschaften von $\bullet$

Wir vermuten, daß

$$[\![C]\!]^\bullet_\rho = [\![C^\bullet]\!]_{\rho_\bullet}$$

gilt, aber der Beweis scheint schwierig. Die folgenden Eigenschaften sind Schritte in Richtung dieser Aussage.

Hilfssatz 5 (Semantische Eigenschaften von $\bullet$).

$$\begin{array}{rcl}
\tau \cong \tau' & \leftrightarrow & \tau^\bullet =_{A_{\mathbf{type}}} \tau^\bullet \\
[\![\tau]\!]_\rho & \cong & [\![\tau]\!]_{\rho_\bullet} \\
[\![\tau]\!]_\rho & \cong & [\![\tau^\bullet]\!]_\rho \\
[\![\tau]\!]^\bullet_\rho & = & [\![\tau^\bullet]\!]_{\rho_\bullet} \\
\pi_1([\![C]\!]_\rho) & \to & \pi_2([\![C]\!]_\rho) \\
\pi_2([\![C]\!]_\rho) & \leftrightarrow & \pi_2([\![C]\!]_{\rho_\bullet}) \\
\pi_2([\![C]\!]_\rho) & \leftrightarrow & \pi_2([\![C^\bullet]\!]_\rho)
\end{array}$$

Beweis:
Wieder sind alle Beweise einfache strukturelle Induktionen. □

Die folgende Proposition zeigt, daß die obige Aussage zumindest für Typgleichungen gilt:

Proposition 8. *Es gilt:*

$$[\![(\tau = \tau')^\bullet]\!]_{\rho_\bullet} = [\![\tau = \tau']\!]^\bullet_\rho.$$

Beweis:
Es gilt

$$[\![(\tau = \tau')^\bullet]\!]_{\rho_\bullet} = [\![\tau^\bullet = \tau'^\bullet]\!]_{\rho_\bullet} = \left(\begin{array}{l} [\![\tau^\bullet]\!]_{\rho_\bullet} = [\![\tau'^\bullet]\!]_{\rho_\bullet} \\ [\![\tau^\bullet]\!]_{\rho_\bullet} \cong [\![\tau'^\bullet]\!]_{\rho_\bullet} \end{array} \right).$$

Auf der anderen Seite haben wir

$$[\![\tau = \tau']\!]^\bullet_\rho = \left(\begin{array}{l} [\![\tau]\!]_\rho \cong [\![\tau']\!]_\rho \\ [\![\tau]\!]_\rho \cong [\![\tau']\!]_\rho \end{array} \right).$$

Die verbleibenden Gleichheiten sehen wir aus den semantische Eigenschaften von $\bullet$. □

Wir vermuten, daß

$$\frac{C \vdash^A C'}{C^\bullet \vdash^A C'^\bullet}$$

gilt, aber wieder scheint der Beweis schwierig. Der folgende Hilfssatz ist ein Schritt in diese Richtung.

Hilfssatz 6. *Es gilt:*

$$\psi^\bullet \tau^\bullet = (\psi\tau)^\bullet.$$

$$\frac{\vdash^A \tau = \tau'}{\vdash^A \tau^\bullet = \tau'^\bullet}.$$

Beweis:
Unter Benutzung von $[\![[\bar{\tau}/\bar{\alpha}]C]\!]_\rho = [\![C]\!]_{\rho[[\![\bar{\tau}]\!]_\rho/\bar{\alpha}]}$ und den semantischen Eigenschaften von $\bullet$ sind die Beweise wieder einfach. □

A.1.1 Subsumption

Die Reflexivität ist leicht zu zeigen

Proposition 9 (Reflexivität der Subsumption). *Es gilt:*

$$C \vdash^A \sigma \sqsubseteq \sigma$$

Beweis:
Sei $\sigma = \forall\bar{\alpha}.\forall\bar{n}.C' \Rightarrow \tau$. Dann schreiben wir die Behauptung als

$$C \wedge^A \forall\bar{\alpha}.\forall\bar{n}.C' \Rightarrow \tau \sqsubseteq \forall\bar{\alpha}'.\forall\bar{n}'.[\bar{\alpha}'/\bar{\alpha}, \bar{n}'/\bar{n}]C' \Rightarrow [\bar{\alpha}'/\bar{\alpha}, \bar{n}'/\bar{n}]\tau.$$

Das Einsetzen der Definition führt zu

$$C \wedge C' \vdash^A \exists\bar{\alpha}'.\exists\bar{n}'.[\bar{\alpha}'/\bar{\alpha}, \bar{n}'/\bar{n}]C' \wedge [\bar{\alpha}'/\bar{\alpha}, \bar{n}'/\bar{n}]\tau = \tau$$

Offensichtlich erfüllen $\bar{\alpha}$ und $\bar{n}$ diese existentielle Bedingung. □

Proposition 10 (Transitivität der Subsumption). *Aus*

$$C \vdash^A \sigma \sqsubseteq \sigma'$$

und

$$C' \vdash^A \sigma' \sqsubseteq \sigma''$$

folgt

$$C \wedge C' \vdash^A \sigma \sqsubseteq \sigma''$$

Beweis:

Wir nehmen an es gelte

$$C_1 \vdash^A \forall \bar{\alpha}.\forall \bar{n}.C \Rightarrow \tau \sqsubseteq \forall \bar{\beta}.\forall \bar{m}.C' \Rightarrow \tau'$$

und

$$C_2 \vdash^A \forall \bar{\beta}.\forall \bar{m}.C' \Rightarrow \tau' \sqsubseteq \forall \bar{\gamma}.\forall \bar{k}.C'' \Rightarrow \tau''.$$

Nach Einsetzen der Definitionen haben wir

$$C_1 \wedge C \vdash^A \exists \bar{\beta}.\exists \bar{m}.(C' \wedge \tau = \tau')$$

und

$$C_2 \wedge C' \vdash^A \exists \bar{\gamma}.\exists \bar{k}.(C'' \wedge \tau' = \tau'').$$

Aus ersterem folgern wir

$$C_1 \wedge C_2 \wedge C \vdash^A \exists \bar{\beta}.\exists \bar{m}.(C_2 \wedge C' \wedge \tau = \tau')$$

und aus der zweiten Gleichung folgt

$$\exists \bar{\beta}.\exists \bar{m}.(C_2 \wedge C' \wedge \tau = \tau') \vdash^A \exists \bar{\beta}.\exists \bar{m}.(\exists \bar{\gamma}.\exists \bar{k}.(C'' \wedge \tau' = \tau'') \wedge \tau = \tau').$$

Diese beiden Aussagen liefern gemeinsam

$$C_1 \wedge C_2 \wedge C \vdash^A \exists \bar{\beta}.\exists \bar{m}.(\exists \bar{\gamma}.\exists \bar{k}.C'' \wedge \tau' = \tau'' \wedge \tau = \tau').$$

Damit gilt auch

$$C_1 \wedge C_2 \wedge C \vdash^A \exists \bar{\beta}.\exists \bar{m}.\exists \bar{\gamma}.\exists \bar{k}.(C'' \wedge \tau = \tau'') \equiv^A \exists \bar{\gamma}.\exists \bar{k}.(C'' \wedge \tau = \tau'').$$

Dies ist aber gerade die Definition der Behauptung. □

A.2 Typinferenz

A.2.1 Eigenschaften des logischen Systems

Proposition 11 (Verschärfung). *Wenn $C' \vdash^A C$ und $C, \Gamma \vdash E : \sigma$ gelten, dann gilt auch $C', \Gamma \vdash E : \sigma$.*

Beweis:
Wir beweisen das durch Induktion über die Derivation von $C, \Gamma \vdash E : \sigma$. Wir betrachten nur nichttriviale Regeln.

- $$(let)\frac{C, \Gamma \vdash E : \sigma \quad C', (\Gamma, x : \sigma) \vdash F : \tau}{C \wedge C', \Gamma \vdash (\textbf{let } x = E \textbf{ in } F) : \tau}$$

  Wir haben $C'' \vdash^A C$ und $C'' \vdash^A C'$ und somit nach Anwendung der (let)-Regel $C'' \wedge C'' \equiv^A C''$ für die Konklusion.

- $$(\forall \Rightarrow I)\frac{C \wedge C', \Gamma \vdash E : \tau}{C \wedge \exists \bar{\alpha}.\exists \bar{n}.C', \Gamma \vdash E : \forall \bar{\alpha}.\forall \bar{n}.C' \Rightarrow \tau} \ \bar{n}, \bar{\alpha} \notin \mathrm{fv}(C, \Gamma)$$

  Wir verwenden $C'' \wedge C' \vdash^A C \wedge C'$ für die Induktionshypothese und haben dann $C'' \wedge \exists \bar{\alpha}.\exists \bar{n}.C' \equiv^A C''$ für die Konklusion.

- $$(\exists I)\frac{C, \Gamma \vdash E : \sigma}{\exists \bar{\alpha}.\exists \bar{n}.C, \Gamma \vdash E : \sigma} \quad \bar{n}, \bar{\alpha} \notin \mathrm{fv}(\Gamma, \sigma)$$

  Wir verwenden $C' \wedge C \vdash^A C'$ für die Induktionshypothese und haben wieder $C' \wedge \exists \bar{\alpha}.\exists \bar{n}.C \equiv^A C'$ für die Konklusion.

Proposition 12 (Konsistenz). *Wenn $C, \Gamma \vdash E : \sigma$ gilt, dann gilt auch $C \wedge \mathrm{ext}(\Gamma) \vdash^A \mathrm{ext}(\sigma)$.*

Beweis:
Wieder verwenden wir eine Induktion über die Derivation von $C, \Gamma \vdash E : \sigma$. Interessant sind hier nur die Regeln, die in der Konklusion allgemeine Typschemata zulassen, da $[\![\mathrm{ext}(\tau)]\!]_\rho$ für jedes τ wahr ist.

- $$(var)\frac{(x:\sigma)\in\Gamma}{C,\Gamma\vdash x:\sigma}$$

  $[\![\mathrm{ext}(\Gamma)]\!]_\rho$ impliziert $[\![\mathrm{ext}(\sigma)]\!]_\rho$.

- $$(fix)\frac{C,(\Gamma,x:\sigma)\vdash E:\sigma \quad C\vdash^A \mathrm{ext}(\sigma)}{C,\Gamma\vdash(\mathbf{fix}\ x::P\Rightarrow\tau\ \mathbf{in}\ E):\sigma}\ \begin{cases}\bar n,\bar\alpha=\mathrm{fv}(P,\tau)\\ \sigma=\forall\bar\alpha.\forall\bar n.P\Rightarrow\tau\end{cases}$$

  $[\![C]\!]_\rho$ impliziert $[\![\mathrm{ext}(\sigma)]\!]_\rho$.

- $$(\forall\Rightarrow I)\frac{C\wedge C',\Gamma\vdash E:\tau}{C\wedge\exists\bar\alpha.\exists\bar n.C',\Gamma\vdash E:\forall\bar\alpha.\forall\bar n.C'\Rightarrow\tau}\ \bar n,\bar\alpha\notin\mathrm{fv}(C,\Gamma)$$

  Offensichtlich gilt $C\wedge\exists\bar\alpha.\exists\bar n.C'\vdash^A\mathrm{ext}(\forall\bar\alpha.\forall\bar n.C'\Rightarrow\tau)$.

- $$(\exists I)\frac{C,\Gamma\vdash E:\sigma}{\exists\bar\alpha.\exists\bar n.C,\Gamma\vdash E:\sigma}\quad\bar n,\bar\alpha\notin\mathrm{fv}(\Gamma,\sigma)$$

  Wir haben
  $$\frac{C\wedge\mathrm{ext}(\Gamma)\vdash^A\mathrm{ext}(\sigma)}{\exists\bar\alpha.\exists\bar n.C\wedge\mathrm{ext}(\Gamma)\vdash^A\mathrm{ext}(\sigma)}.$$

- $$(=)\frac{C,[\bar\tau/\bar\alpha][\bar t/\bar n]\Gamma\vdash E:[\bar\tau/\bar\alpha][\bar t/\bar n]\sigma \quad C\vdash^A\bar t=\bar t',\bar\tau=\bar\tau'}{C,[\bar\tau'/\bar\alpha][\bar t'/\bar n]\Gamma\vdash E:[\bar\tau'/\bar\alpha][\bar t'/\bar n]\sigma}$$

  Wir haben
  $$\frac{\dfrac{\dfrac{C\vdash^A\mathrm{ext}([\bar\tau/\bar\alpha][\bar t/\bar n]\sigma)}{C\vdash^A[\bar\tau/\bar\alpha][\bar t/\bar n]\,\mathrm{ext}(\sigma)}}{C\vdash^A[\bar\tau'/\bar\alpha][\bar t'/\bar n]\,\mathrm{ext}(\sigma)}}{C\vdash^A\mathrm{ext}([\bar\tau'/\bar\alpha][\bar t'/\bar n]\sigma)}\ .$$

- $$(\mathbf{data}\ I)\frac{\mathtt{data}\ \mathtt{T}\bar\alpha\bar n=\sum_i D_i\bar\tau_i\Leftarrow Q_i,\bar\beta_i\bar m_i \quad C\vdash^A\exists\bar m_i.\exists\bar n.Q_i}{C,\Gamma\vdash D_i:\forall\bar m_i.\forall\bar n.\forall\bar\beta_i.\forall\bar\alpha.Q_i\Rightarrow(\bar\tau_i\to T\bar\alpha\bar n)}$$

  Es gilt $\mathrm{ext}(\forall\bar m_i.\forall\bar n.\forall\bar\beta_i.\forall\bar\alpha.Q_i\Rightarrow(\bar\tau_i\to T\bar\alpha\bar n))\equiv^A\exists\bar m_i.\exists\bar n.Q_i$.

□

A.2.2 Eigenschaften des deterministischen Systems

Proposition 13 (Kontextabschwächung). *Wenn* $C'' \vdash^A \Gamma \sqsubseteq \Gamma'$ *und* $C, \Gamma \vdash^W E : \alpha$ *gelten, dann gilt auch* $C', \Gamma' \vdash^W E : \alpha$, *wobei* $C'' \wedge C \vdash^A C'$.

Beweis:

Wir zeigen das mit Induktion über die $\vdash^W$ Ableitung. Wir werden die letzte Regel betrachten und die Induktionshypothese auf den Antezedent anwenden. Danach wenden wir die letzte Regel für Γ' statt für Γ an. Am Ende müssen wir jeweils $C'' \wedge C \vdash^A C'$ aus den entsprechenden Aussagen für die Induktionshypothese und der Allgemeinheitsbedingung für Γ und Γ' herleiten.

- $$(var^W)\frac{(x : \forall\bar{\alpha}.\forall\bar{n}.C \Rightarrow \tau) \in \Gamma}{\exists\bar{\alpha}.\exists\bar{n}.(C \wedge \delta = \tau), \Gamma \vdash^W x : \delta} \quad \delta\, new$$

  Aus der Hypothese wissen wir, daß $(x : \forall\bar{\beta}.\forall\bar{m}.C' \Rightarrow \tau') \in \Gamma'$. Wir wenden die entsprechende Regel an:

  $$\frac{(x : \forall\bar{\beta}.\forall\bar{m}.C' \Rightarrow \tau') \in \Gamma'}{\exists\bar{\beta}.\exists\bar{m}.(C' \wedge \delta = \tau'), \Gamma' \vdash^W x : \delta}$$

  Wir zeigen es hier ausführlich (später knapper):

  $$\dfrac{\dfrac{\dfrac{\dfrac{C'' \vdash^A \forall\bar{\alpha}.\forall\bar{n}.C \Rightarrow \tau \sqsubseteq \forall\bar{\beta}.\forall\bar{m}.C' \Rightarrow \tau'}{C'' \wedge C \vdash^A \exists\bar{\beta}.\exists\bar{m}.C' \wedge \tau = \tau'}}{C'' \wedge C \wedge \delta = \tau \vdash^A \exists\bar{\beta}.\exists\bar{m}.C' \wedge \tau = \tau'}}{C'' \wedge C \wedge \delta = \tau \vdash^A \exists\bar{\beta}.\exists\bar{m}.C' \wedge \delta = \tau'}}{C'' \wedge \exists\bar{\alpha}.\exists\bar{n}.(C \wedge \delta = \tau) \vdash^A \exists\bar{\beta}.\exists\bar{m}.C' \wedge \delta = \tau'}$$

- $$(\rightarrow I^W)\frac{C, (\Gamma, x : \alpha) \vdash^W E : \beta}{\exists\alpha.\exists\beta.(\delta = \alpha \rightarrow \beta \wedge C), \Gamma \vdash^W \lambda x.E : \delta} \quad \alpha, \delta\, new$$

Wir wissen also $C'' \vdash^A (\Gamma, x:\alpha) \sqsubseteq (\Gamma', x:\alpha)$. Und mit der Induktionshypothese können wir die folgende Regel anwenden:

$$\frac{C', (\Gamma', x:\alpha) \vdash^W E : \beta}{\exists\alpha.\exists\beta.(\delta = \alpha \to \beta \wedge C'), \Gamma' \vdash^W \lambda x.E : \delta}$$

Wir zeigen:

$$\frac{\dfrac{C'' \wedge C \vdash^A C'}{C'' \wedge C \wedge \delta = \alpha \to \beta \vdash^A C' \wedge \delta = \alpha \to \beta}}{C'' \wedge \exists\alpha.\exists\beta.(\delta = \alpha \to \beta \wedge C) \vdash^A \exists\alpha.\exists\beta.(\delta = \alpha \to \beta \wedge C')}$$

- $$(\to E^W)\frac{C, \Gamma \vdash^W E : \alpha \quad C', \Gamma \vdash^W F : \beta}{\exists\alpha.\exists\beta.(C \wedge C' \wedge \alpha = \beta \to \delta), \Gamma \vdash^W EF : \delta} \quad \delta\, new$$

  Wir verwenden die Hypothese zweimal und wenden die Regel an.

  $$\frac{C'''', \Gamma' \vdash^W E : \alpha \quad C''', \Gamma' \vdash^W F : \beta}{\exists\alpha.\exists\beta.(C'''' \wedge C''' \wedge \alpha = \beta \to \delta), \Gamma' \vdash^W EF : \delta}$$

  Nun müssen wir noch die Bedingung herleiten.

  $$\frac{\dfrac{C'' \wedge C \vdash^A C'''' \quad C'' \wedge C' \vdash^A C'''}{C'' \wedge C \wedge C' \wedge \alpha = \beta \to \delta \vdash^A C'''' \wedge C''' \wedge \alpha = \beta \to \delta}}{\begin{array}{c} C'' \wedge \exists\alpha.\exists\beta.(C \wedge C' \wedge \alpha = \beta \to \delta) \\ \vdash^A \exists\alpha.\exists\beta.(C'''' \wedge C''' \wedge \alpha = \beta \to \delta) \end{array}}$$

- $$(let^W)\frac{\begin{array}{c} C, \Gamma \vdash^W E : \alpha \\ C', (\Gamma, x : \forall\alpha.C \Rightarrow \alpha) \vdash^W F : \beta \end{array}}{C' \wedge \exists\alpha.C, \Gamma \vdash^W (\textbf{let } x = E \textbf{ in } F) : \beta}$$

  Diese Regel ist etwas komplizierter, da wir nur eine Induktionshypothese sofort anwenden können. Wir haben $C''', \Gamma' \vdash^W E : \alpha$ mit $C'' \wedge C \vdash^A C'''$. Aber nun gilt:

  $$\frac{\dfrac{\dfrac{C'' \wedge C \vdash^A C''' \quad C''' \equiv^A \exists\beta.\beta = \alpha \wedge [\beta/\alpha]C'''}{C'' \wedge C \vdash^A \exists\beta.\beta = \alpha \wedge [\beta/\alpha]C'''}}{C'' \vdash^A (\Gamma', x : \forall\alpha.C \Rightarrow \alpha) \sqsubseteq (\Gamma', x : \forall\beta.[\beta/\alpha]C''' \Rightarrow \beta)}}{C'' \vdash^A (\Gamma', x : \forall\alpha.C \Rightarrow \alpha) \sqsubseteq (\Gamma', x : \forall\alpha.C''' \Rightarrow \alpha)}$$

Jetzt können wir die Induktionshypothese zum zweiten Mal anwenden:

$$\frac{C''', \Gamma' \vdash^W E : \alpha \quad C'''', (\Gamma', x : \forall\alpha.C''' \Rightarrow \alpha) \vdash^W F : \beta}{C'''' \wedge \exists\alpha.C''', \Gamma \vdash^W (\textbf{let } x = E \textbf{ in } F) : \beta}$$

Die Bedingung, die wir zeigen müssen ist diesmal einfach:

$$\frac{C'' \wedge C \vdash^A C''' \quad C'' \wedge C' \vdash^A C''''}{C'' \wedge C' \wedge \exists\alpha.C \vdash^A C'''' \wedge \exists\alpha.C'''}$$

- $$(fix^W)\frac{C, (\Gamma, x : \forall\bar{\alpha}.\forall\bar{n}.P \Rightarrow \tau) \vdash^W E : \gamma}{\begin{array}{l}\exists\bar{\alpha}.\exists\bar{n}.(\delta = \tau \wedge P \wedge \\ \quad \forall\bar{\alpha}.\forall\bar{n}.(P \rightarrow \exists\gamma.(C \wedge \tau = \gamma))), \\ \quad \Gamma \vdash^W (\textbf{fix } x :: P \Rightarrow \tau \textbf{ in } E) : \delta\end{array}} \left\{\begin{array}{l}\delta \, new \\ \bar{\alpha}, \bar{n} = \mathrm{fv}(P, \tau)\end{array}\right.$$

  Mit der Induktionshypothese haben wir:

$$\frac{\begin{array}{c}C', (\Gamma', x : \forall\bar{\alpha}.\forall\bar{n}.P \Rightarrow \tau) \vdash^W E : \gamma\end{array}}{\begin{array}{l}\exists\bar{\alpha}.\exists\bar{n}.(\delta = \tau \wedge P \wedge \forall\bar{\alpha}.\forall\bar{n}.(P \rightarrow \exists\gamma.(C' \wedge \tau = \gamma))), \\ \quad \Gamma' \vdash^W (\textbf{fix } x :: \forall\bar{\alpha}.\forall\bar{n}.P \Rightarrow \tau \textbf{ in } E) : \delta\end{array}}$$

  Wir leiten damit

$$\frac{\begin{array}{c}C'' \wedge C \vdash^A C'\end{array}}{\begin{array}{c}P \rightarrow C'' \wedge P \rightarrow \exists\gamma.(C \wedge \tau = \gamma) \vdash^A P \rightarrow \exists\gamma.(C' \wedge \tau = \gamma)\end{array}}$$
$$\frac{\begin{array}{c}C'' \wedge P \rightarrow \exists\gamma.(C \wedge \tau = \gamma) \vdash^A P \rightarrow \exists\gamma.(C' \wedge \tau = \gamma)\end{array}}{\begin{array}{l}C'' \wedge \exists\bar{\alpha}.\exists\bar{n}.(\delta = \tau \wedge P \wedge \forall\bar{\alpha}.\forall\bar{n}.(P \rightarrow \exists\gamma.(C \wedge \tau = \gamma))) \\ \quad \vdash^A \exists\bar{\alpha}.\exists\bar{n}.(\delta = \tau \wedge P \wedge \forall\bar{\alpha}.\forall\bar{n}.(P \rightarrow \exists\gamma.(C' \wedge \tau = \gamma)))\end{array}}$$

  her.

- $$(\textbf{data } I^W)\frac{\texttt{data T}\bar{\alpha}\bar{n} = \sum_i D_i\bar{\tau}_i \Leftarrow Q_i, \bar{\beta}_i\bar{m}_i}{\begin{array}{l}\exists\bar{\alpha}.\exists\bar{\beta}_i.\exists\bar{n}.\exists\bar{m}_i.(\delta = \bar{\tau}_i \rightarrow T\bar{\alpha}\bar{n} \wedge Q_i), \\ \quad \Gamma \vdash^W D_i : \delta\end{array}} \quad \delta \, new$$

  Der Antezedent hängt nicht von Γ ab, und so gilt die Behauptung offensichtlich mit $C' = C$.

- $$(\textbf{data } E^W)\frac{\begin{array}{l}\texttt{data T}\bar\alpha\bar n = \sum_i D_i\bar\tau_i \Leftarrow Q_i, \bar\beta_i\bar m_i\\ C, \Gamma \vdash^W F : \eta\\ C_i, (\Gamma, \bar y_i : \bar\rho_i) \vdash^W E_i : \zeta_i\end{array}}{\begin{array}{l}\exists\eta.\exists\bar\alpha.\exists\bar n.(\eta = T\bar\alpha\bar n \wedge C \wedge\\ \quad \forall i.\forall\bar\beta_i.\forall\bar m_i.\exists\zeta_i.\exists\bar\rho_i.(Q_i \rightarrow\\ \quad (\bar\rho_i = \bar\tau_i \wedge C_i \wedge \zeta_i = \delta))),\\ \quad \Gamma \vdash^W \textbf{case } F \textbf{ of } \{D_i\bar y_i \Rightarrow E_i\} : \delta\end{array}} \quad \delta, \bar\rho_i \, new$$

  Mit der Induktionshypothese bekommen wir

  $$\frac{C', \Gamma' \vdash^W x : \eta \quad C_i', (\Gamma', \bar y_i : \bar\rho_i) \vdash^W E_i : \zeta_i}{\begin{array}{l}\exists\eta.\exists\bar\alpha.\exists\bar n.(\eta = T\bar\alpha\bar n \wedge C' \wedge\\ \quad \forall i.\forall\bar\beta_i\forall\bar m_i.\exists\zeta_i.\exists\bar\rho_i.(Q_i \rightarrow (\bar\rho_i = \bar\tau_i \wedge C_i' \wedge \zeta_i = \delta))),\\ \quad \Gamma' \vdash^W \textbf{case } x \textbf{ of } \{D_i\bar y_i \Rightarrow E_i\} : \delta\end{array}},$$

  und mit der vertrauten Herleitung schließt der Beweis.

  $$\frac{C'' \wedge C \vdash^A C' \qquad \dfrac{C'' \wedge C_i \vdash^A C_i'}{\begin{array}{l}C'' \wedge \forall\bar\beta_i.\forall\bar m_i.\exists\zeta_i.\exists\bar\rho_i.\\ \quad (Q_i \rightarrow (\bar\rho_i = \bar\tau_i \wedge C_i \wedge \zeta_i = \delta))\\ \vdash^A \forall\bar\beta_i.\forall\bar m_i.\exists\zeta_i.\exists\bar\rho_i.\\ \quad (Q_i \rightarrow (\bar\rho_i = \bar\tau_i \wedge C_i' \wedge \zeta_i = \delta))\end{array}}}{\begin{array}{l}C'' \wedge \exists\eta.\exists\bar\alpha.\exists\bar n.(\eta = T\bar\alpha\bar n \wedge C \wedge\\ \quad \forall i.\forall\bar\beta_i.\forall\bar m_i.\exists\zeta_i.\exists\bar\rho_i.(Q_i \rightarrow (\bar\rho_i = \bar\tau_i \wedge C_i \wedge \zeta_i = \delta)))\\ \vdash^A \exists\eta.\exists\bar\alpha.\exists\bar n.(\eta = T\bar\alpha\bar n \wedge C' \wedge\\ \quad \forall i.\forall\bar\beta_i.\forall\bar m_i.\exists\zeta_i.\exists\bar\rho_i.(Q_i \rightarrow (\bar\rho_i = \bar\tau_i \wedge C_i' \wedge \zeta_i = \delta)))\end{array}}$$

□

A.2.3 Korrektheit

Satz 2 (Korrektheit). *Wenn* $C, \Gamma \vdash^W E : \alpha$ *gilt, dann gilt auch* $C, \Gamma \vdash E : \alpha$.

Beweis:
Wir zeigen, daß wir jede der $\vdash^W$-Regeln mit $\vdash$-Regeln simulieren können. Wir für jede $\vdash^W$-Regel eine $\vdash$-Ableitung an.

- $$(var^W)\frac{(x : \forall\bar{\alpha}.\forall\bar{n}.C \Rightarrow \tau) \in \Gamma}{\exists\bar{\alpha}.\exists\bar{n}.(C \wedge \delta = \tau), \Gamma \vdash^W x : \delta} \quad \delta\, new$$

  Wir haben die Ableitung:

  $$\dfrac{\dfrac{\dfrac{\dfrac{(x : \forall\bar{\alpha}.\forall\bar{n}.C \Rightarrow \tau) \in \Gamma}{C, \Gamma \vdash x : \forall\bar{\alpha}.\forall\bar{n}.C \Rightarrow \tau} \quad C \vdash^A C}{C, \Gamma \vdash x : \tau}}{C \wedge \delta = \tau, \Gamma \vdash x : \delta}}{\exists\bar{\alpha}.\exists\bar{n}.(C \wedge \delta = \tau), \Gamma \vdash x : \delta}$$

- $$(\rightarrow I^W)\frac{C, (\Gamma, x : \alpha) \vdash^W E : \beta}{\exists\alpha.\exists\beta.(\delta = \alpha \rightarrow \beta \wedge C), \Gamma \vdash^W \lambda x.E : \delta} \quad \alpha, \delta\, new$$

  Wir haben die Ableitung:

  $$\dfrac{\dfrac{\dfrac{C, (\Gamma, x : \alpha) \vdash E : \beta}{C, \Gamma \vdash \lambda x.E : \alpha \rightarrow \beta}}{\delta = \alpha \rightarrow \beta \wedge C, \Gamma \vdash \lambda x.E : \delta}}{\exists\alpha.\exists\beta.(\delta = \alpha \rightarrow \beta \wedge C), \Gamma \vdash \lambda x.E : \delta}$$

- $$(\rightarrow E^W)\frac{C, \Gamma \vdash^W E : \alpha \quad C', \Gamma \vdash^W F : \beta}{\exists\alpha.\exists\beta.(C \wedge C' \wedge \alpha = \beta \rightarrow \delta), \Gamma \vdash^W EF : \delta} \quad \delta\, new$$

Wir haben die Ableitung:

$$\dfrac{\dfrac{C, \Gamma \vdash E : \alpha}{C \wedge \alpha = \beta \to \delta, \Gamma \vdash E : \beta \to \delta} \quad C', \Gamma \vdash F : \beta}{C \wedge C' \wedge \alpha = \beta \to \delta, \Gamma \vdash EF : \delta}$$
$$\overline{\exists\alpha.\exists\beta.(C \wedge C' \wedge \alpha = \beta \to \delta), \Gamma \vdash EF : \delta}$$

- $$(let^W)\dfrac{\begin{array}{c} C, \Gamma \vdash^W E : \alpha \\ C', (\Gamma, x : \forall\alpha.C \Rightarrow \alpha) \vdash^W F : \beta \end{array}}{C' \wedge \exists\alpha.C, \Gamma \vdash^W (\textbf{let } x = E \textbf{ in } F) : \beta}$$

Wir haben die Ableitung:

$$\dfrac{\dfrac{C, \Gamma \vdash E : \alpha}{\exists\alpha.C, \Gamma \vdash E : \forall\alpha.C \Rightarrow \alpha} \quad C', (\Gamma, x : \forall\alpha.C \Rightarrow \alpha) \vdash F : \beta}{C' \wedge \exists\alpha.C, \Gamma \vdash (\textbf{let } x = E \textbf{ in } F) : \beta}$$

- $$(fix^W)\dfrac{\begin{array}{l} C, (\Gamma, x : \forall\bar{\alpha}.\forall\bar{n}.P \Rightarrow \tau) \vdash^W E : \gamma \\ \exists\bar{\alpha}.\exists\bar{n}.(\delta = \tau \wedge P \wedge \\ \quad \forall\bar{\alpha}.\forall\bar{n}.(P \to \exists\gamma.(C \wedge \tau = \gamma))), \\ \quad \Gamma \vdash^W (\textbf{fix } x :: P \Rightarrow \tau \textbf{ in } E) : \delta \end{array}}{\left\{ \begin{array}{l} \delta\, new \\ \bar{\alpha}, \bar{n} = \mathrm{fv}(P, \tau) \end{array} \right.}$$

Wir haben die Ableitung:

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
C, (\Gamma, x : \forall\bar{\alpha}.\forall\bar{n}.P \Rightarrow \tau) \vdash E : \gamma
}{
C \wedge \gamma = \tau, (\Gamma, x : \forall\bar{\alpha}.\forall\bar{n}.P \Rightarrow \tau) \vdash E : \tau
}(\exists I)
}{
\exists\gamma(C \wedge \gamma = \tau),\ (\Gamma, x : \forall\bar{\alpha}.\forall\bar{n}.P \Rightarrow \tau) \vdash E : \tau
}
}{
P \wedge \forall\bar{\alpha}.\forall\bar{n}.(P \rightarrow \exists\gamma.(\gamma = \tau \wedge C)),\ (\Gamma, x : \forall\bar{\alpha}.\forall\bar{n}.P \Rightarrow \tau) \vdash E : \tau
}(\forall \Rightarrow I)
}{
(\exists\bar{\alpha}.\exists\bar{n}.P) \wedge \forall\bar{\alpha}.\forall\bar{n}.\exists\gamma.(P \rightarrow \exists\gamma.(\gamma = \tau \wedge C)),\ (\Gamma, x : \forall\bar{\alpha}.\forall\bar{n}.P \Rightarrow \tau) \vdash E : \forall\bar{\alpha}.\forall\bar{n}.P \Rightarrow \tau
}(fix)
}{
(\exists\bar{\alpha}.\exists\bar{n}.P) \wedge \forall\bar{\alpha}.\forall\bar{n}.\ (P \rightarrow \exists\gamma.(\gamma = \tau \wedge C)),\ (\Gamma, x : \forall\bar{\alpha}.\forall\bar{n}.P \Rightarrow \tau) \vdash (\mathbf{fix}\ x :: P \Rightarrow \tau\ \mathbf{in}\ E) : \forall\bar{\alpha}.\forall\bar{n}.P \Rightarrow \tau
}
}{
P \wedge \forall\bar{\alpha}.\forall\bar{n}.(P \rightarrow \exists\gamma.(\gamma = \tau \wedge C)),\ (\Gamma, x : \forall\bar{\alpha}.\forall\bar{n}.P \Rightarrow \tau) \vdash (\mathbf{fix}\ x :: P \Rightarrow \tau\ \mathbf{in}\ E) : \forall\bar{\alpha}.\forall\bar{n}.P \Rightarrow \tau
}(\forall \Rightarrow E)
}{
P \wedge \forall\bar{\alpha}.\forall\bar{n}.(P \rightarrow (\gamma = \tau \wedge C)),\ (\Gamma, x : \forall\bar{\alpha}.\forall\bar{n}.P \Rightarrow \tau) \vdash (\mathbf{fix}\ x :: P \Rightarrow \tau\ \mathbf{in}\ E) : \tau
}
}{
\delta = \tau \wedge P \wedge \forall\bar{\alpha}.\forall\bar{n}.(P \rightarrow \exists\gamma.(\gamma = \tau \wedge C)),\ (\Gamma, x : \forall\bar{\alpha}.\forall\bar{n}.P \Rightarrow \tau) \vdash (\mathbf{fix}\ x :: P \Rightarrow \tau\ \mathbf{in}\ E) : \delta
}(\exists I)
}{
\exists\bar{\alpha}.\exists\bar{\beta}.(\delta = \tau \wedge P \wedge \forall\bar{\alpha}.\forall\bar{n}.(P \rightarrow \exists\gamma.(\gamma = \tau \wedge C))),\ (\Gamma, x : \forall\bar{\alpha}.\forall\bar{n}.P \Rightarrow \tau) \vdash (\mathbf{fix}\ x :: P \Rightarrow \tau\ \mathbf{in}\ E) : \delta
}(\exists I)
$$

Die weiteren Bedingungen für (fix) und $(\forall \Rightarrow E)$ sind jeweils trivialerweise erfüllt.

- $$
\dfrac{
\texttt{data T}\bar{\alpha}\bar{n} = \sum_i D_i\bar{\tau}_i \Leftarrow Q_i, \bar{\beta}_i\bar{m}_i
}{
\exists\bar{\alpha}.\exists\bar{\beta}_i.\exists\bar{n}.\exists\bar{m}_i.(\delta = \bar{\tau}_i \rightarrow T\bar{\alpha}\bar{n} \wedge Q_i),\ \Gamma \vdash^W D_i : \delta
}(\mathbf{data}\ I^W)\quad \delta\, new
$$

Wir haben die Ableitung:

$$\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{l}\texttt{data}\ T\bar{\alpha}\bar{n} = \sum_i D_i\bar{\tau}_i \Leftarrow Q_i, \bar{\beta}_i\bar{m}_i \\ Q_i \vdash^A \exists\bar{n}.\exists\bar{m}_i.Q_i\end{array}}{Q_i, \Gamma \vdash D_i : \forall\bar{m}_i.\forall\bar{n}.\forall\bar{\beta}_i.\forall\bar{\alpha}.Q_i \Rightarrow (\bar{\tau}_i \to T\bar{\alpha}\bar{n})}}{Q_i, \Gamma \vdash D_i : \bar{\tau}_i \to T\bar{\alpha}\bar{n}}}{\delta = \bar{\tau}_i \to T\bar{\alpha}\bar{n} \wedge Q_i, \Gamma \vdash D_i : \delta}}{\exists\bar{\alpha}.\exists\bar{\beta}_i.\exists\bar{n}.\exists\bar{m}_i.(\delta = \bar{\tau}_i \to T\bar{\alpha}\bar{n} \wedge Q_i), \Gamma \vdash D_i : \delta}$$

- $$(\textbf{data}\ E^W)\dfrac{\begin{array}{l}\texttt{data T}\bar{\alpha}\bar{n} = \sum_i D_i\bar{\tau}_i \Leftarrow Q_i, \bar{\beta}_i\bar{m}_i \\ C, \Gamma \vdash^W F : \eta \\ C_i, (\Gamma, \bar{y}_i : \bar{\rho}_i) \vdash^W E_i : \zeta_i\end{array}}{\begin{array}{l}\exists\eta.\exists\bar{\alpha}.\exists\bar{n}.(\eta = T\bar{\alpha}\bar{n} \wedge C \wedge \\ \quad \forall i.\forall\bar{\beta}_i.\forall\bar{m}_i.\exists\zeta_i.\exists\bar{\rho}_i.(Q_i \to \\ \quad (\bar{\rho}_i = \bar{\tau}_i \wedge C_i \wedge \zeta_i = \delta))), \\ \quad \Gamma \vdash^W \textbf{case}\ F\ \textbf{of}\ \{D_i\bar{y}_i \Rightarrow E_i\} : \delta\end{array}} \quad \delta, \bar{\rho}_i\, new$$

  Wir leiten zuerst

  $$\dfrac{C, \Gamma \vdash F : \eta}{\eta = T\bar{\alpha}\bar{n} \wedge C, \Gamma \vdash F : T\bar{\alpha}\bar{n}}$$

  und

  $$\dfrac{\dfrac{\dfrac{\dfrac{C_i, (\Gamma, \bar{y}_i : \bar{\rho}_i) \vdash E_i : \zeta_i}{\exists\zeta_i.\exists\bar{\rho}_i.\bar{\tau}_i = \bar{\rho}_i \wedge \delta = \zeta_i \wedge C_i, (\Gamma, \bar{y}_i : \bar{\tau}_i) \vdash E_i : \delta}}{\begin{array}{c}Q_i \wedge \exists\zeta_i.\exists\bar{\rho}_i.(Q_i \to (\bar{\tau}_i = \bar{\rho}_i \wedge \delta = \zeta_i \wedge C_i)), \\ (\Gamma, \bar{y}_i : \bar{\tau}_i) \vdash E_i : \delta\end{array}}}{\begin{array}{l}Q_i \wedge \forall i.\forall\bar{\gamma}_i.\forall\bar{m}_i.\exists\zeta_i.\exists\bar{\rho}_i.(Q_i \to (\bar{\tau}_i = \bar{\rho}_i \wedge \delta = \zeta_i \wedge C_i)), \\ (\Gamma, \bar{y}_i : \bar{\tau}_i) \vdash E_i : \delta\end{array}}$$

ab. Mit $\texttt{data}\ T\bar{\alpha}\bar{n} = \sum_i D_i\bar{\tau}_i \Leftarrow Q_i$ haben wir dann die Ableitung:

$$\dfrac{\begin{array}{ll}\eta = T\bar{\alpha}\bar{n} \wedge C, & Q_i \wedge \forall i.\forall\bar{\gamma}_i.\forall\bar{m}_i.\exists\zeta_i.\exists\bar{\rho}_i. \\ \Gamma \vdash F : T\bar{\alpha}\bar{n} & \quad (Q_i \to (\bar{\tau}_i = \bar{\rho}_i \wedge \delta = \zeta_i \wedge C_i)), \\ & \quad (\Gamma, \bar{y}_i : \bar{\tau}_i) \vdash E_i : \delta\end{array}}{\begin{array}{c}(\eta = T\bar{\alpha}\bar{n} \wedge C \wedge \forall i.\forall\bar{\gamma}_i.\forall\bar{m}_i.\exists\zeta_i.\exists\bar{\rho}_i. \\ (Q_i \to (\bar{\rho}_i = \bar{\tau}_i \wedge C_i \wedge \zeta_i = \delta))), \\ \Gamma \vdash \textbf{case}\ F\ \textbf{of}\ \{D_i\bar{y}_i \Rightarrow E_i\} : \delta\end{array}}$$

$$\dfrac{\begin{array}{c}(\eta = T\bar{\alpha}\bar{n} \wedge C \wedge \forall i.\forall\bar{\gamma}_i.\forall\bar{m}_i.\exists\zeta_i.\exists\bar{\rho}_i. \\ (Q_i \to (\bar{\rho}_i = \bar{\tau}_i \wedge C_i \wedge \zeta_i = \delta))), \\ \Gamma \vdash \textbf{case}\ F\ \textbf{of}\ \{D_i\bar{y}_i \Rightarrow E_i\} : \delta\end{array}}{\begin{array}{c}\exists\eta.\exists\bar{\alpha}.\exists\bar{n}.(\eta = T\bar{\alpha}\bar{n} \wedge C \wedge \forall i.\forall\bar{\gamma}_i.\forall\bar{m}_i.\exists\zeta_i.\exists\bar{\rho}_i. \\ (Q_i \to (\bar{\rho}_i = \bar{\tau}_i \wedge C_i \wedge \zeta_i = \delta))), \\ \Gamma \vdash \textbf{case}\ F\ \textbf{of}\ \{D_i\bar{y}_i \Rightarrow E_i\} : \delta\end{array}}$$

□

A.2.4 Vollständigkeit

Satz 3 (Vollständigkeit). *Wenn $C, \Gamma \vdash E : \forall\bar{\alpha}.\forall\bar{n}.C' \Rightarrow \tau$ gilt, dann existiert eine Ableitung $C'', \Gamma \vdash^W E : \beta$ mit $C \wedge C' \wedge \mathrm{ext}(\Gamma) \vdash^A \exists\beta.(C'' \wedge \beta = \tau)$.*

Beweis:
Wir zeigen, daß wir für jede Ableitung in $\vdash$ eine allgemeinere $\vdash^W$ Ableitung finden können. Wir verwenden dazu eine Induktion über Ableitungen in $\vdash$. Wir wissen aus der Induktionshypothese, daß für den Antezedent eine $\vdash^W$ Aussage existiert. Wir wenden die der letzten $\vdash$-Regel entsprechende $\vdash^W$-Regel an, wenn es eine solche gibt. Am Ende müssen wir dann die Allgemeinheitsbedingung aus der Allgemeinheitsbedingung der Induktionshypothese herleiten.

- $$(var)\dfrac{(x : \sigma) \in \Gamma}{C, \Gamma \vdash x : \sigma}$$

  Bei dieser Regel brauchen wir die Induktionshypothese nicht, wir setzen sofort die $\vdash^W$ Ableitung an.

  Sei $\sigma = \forall\bar{\alpha}.\forall\bar{n}.C' \Rightarrow \tau$.

  $$\dfrac{(x : \sigma) \in \Gamma}{\exists\bar{\alpha}.\exists\bar{n}.(C' \wedge \delta = \tau), \Gamma \vdash^W x : \delta}$$

Wir müssen jetzt noch zeigen, daß

$$C \wedge C' \wedge \mathrm{ext}(\Gamma) \vdash^A \exists\delta.(\delta = \tau \wedge \exists\bar{\alpha}.\exists\bar{n}.(C' \wedge \delta = \tau))$$

Wir setzen auf der rechten Seite $\delta = \tau, \bar{\alpha} = \bar{\alpha}$ und $\bar{n} = \bar{n}$. Nun ist die Aussage trivial.

- $$(\to I)\frac{C, (\Gamma, x : \tau) \vdash E : \tau'}{C, \Gamma \vdash \lambda x.E : \tau \to \tau'}$$

  Wir haben $C', (\Gamma, x : \tau) \vdash^W E : \beta$, und weil $\alpha = \tau \vdash^A (\Gamma, x : \tau) \sqsubseteq (\Gamma, x : \alpha)$ gilt, bekommen wir

  $$\frac{C''', (\Gamma, x : \alpha) \vdash^W E : \beta}{\exists\alpha.\exists\beta.(\delta = \alpha \to \beta \wedge C'''), \Gamma \vdash^W \lambda x.E : \delta},$$

  wobei $C' \wedge \alpha = \tau \vdash^A C'''$. Es reicht nun,

  $$\frac{C \wedge \mathrm{ext}(\Gamma) \vdash^A \exists\beta.C' \wedge \beta = \tau' \quad C' \wedge \alpha = \tau \vdash^A C'''}{C \wedge \mathrm{ext}(\Gamma) \vdash^A \exists\delta.(\delta = \tau \to \tau' \wedge \exists\alpha.\exists\beta.(\delta = \alpha \to \beta \wedge C'''))}$$

  zu zeigen. Dazu setzen wir $\delta = \tau \to \tau'$.

  $$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{C' \wedge \alpha = \tau \vdash^A C'''}{C' \wedge \alpha = \tau \vdash^A C''' \wedge \alpha = \tau}}{C' \vdash^A \exists\alpha.C''' \wedge \alpha = \tau}}{C' \wedge \beta = \tau' \vdash^A \exists\alpha.C''' \wedge \alpha = \tau \wedge \beta = \tau'}}{\exists\beta.C' \wedge \beta = \tau' \vdash^A \exists\beta.\exists\alpha.C''' \wedge \alpha \to \beta = \tau \to \tau'}$$

  Mit der anderen Voraussetzung $C \wedge \mathrm{ext}(\Gamma) \vdash^A \exists\beta.C' \wedge \beta = \tau'$ und der Transitivität von $\vdash^A$ ergibt sich die geforderte Formel.

- $$(\to E)\frac{C, \Gamma \vdash E : \tau' \to \tau \quad C, \Gamma \vdash F : \tau'}{C, \Gamma \vdash EF : \tau}$$

  Mit der Induktionshypothese haben wir die $\vdash^W$ Ableitung

  $$\frac{C', \Gamma \vdash^W E : \alpha \quad C'', \Gamma \vdash^W F : \beta}{\exists\alpha.\exists\beta.(C' \wedge C'' \wedge \alpha = \beta \to \delta), \Gamma \vdash^W EF : \delta}.$$

Wir zeigen

$$\frac{\begin{array}{c} C \wedge \mathrm{ext}(\Gamma) \vdash^A \exists\alpha.(C' \wedge \alpha = \tau' \to \tau) \\ C \wedge \mathrm{ext}(\Gamma) \vdash^A \exists\beta.(C'' \wedge \beta = \tau') \end{array}}{C \wedge \mathrm{ext}(\Gamma) \vdash^A \exists\delta.(\exists\alpha.\exists\beta.(C' \wedge C'' \wedge \alpha = \beta \to \delta)) \wedge \delta = \tau},$$

indem wir $\delta = \tau$ setzen. Die beiden Voraussetzungen können wir zu

$$C \wedge \mathrm{ext}(\Gamma) \vdash^A \exists\alpha.\exists\beta.(C' \wedge C'' \wedge \alpha = \tau' \to \tau \wedge \beta = \tau')$$

zusammenfassen, was die Behauptung impliziert.

- $$(let)\frac{C, \Gamma \vdash E : \sigma \quad C', (\Gamma, x : \sigma) \vdash F : \tau}{C \wedge C', \Gamma \vdash (\mathbf{let}\ x = E\ \mathbf{in}\ F) : \tau}$$

  Mit Induktion haben wir eine $\vdash^W$ Ableitung $C'', \Gamma \vdash^W E : \alpha$. Wir schreiben $\sigma = \forall\bar{\gamma}.\forall\bar{m}.C_\sigma \Rightarrow \tau'$ und folgern:

  $$\frac{C \wedge C_\sigma \wedge \mathrm{ext}(\Gamma) \vdash^A \exists\alpha.C'' \wedge \alpha = \tau'}{C \wedge \mathrm{ext}(\Gamma) \vdash^A \sigma \sqsubseteq \forall\alpha.C'' \Rightarrow \alpha}$$

  Jetzt können wir zum zweiten Mal die Induktionshypothese anwenden und, nach Kontextabschwächung, die (let^W) Regel anwenden.

  $$\frac{C'', \Gamma \vdash^W E : \alpha \qquad \dfrac{C''', (\Gamma, x : \sigma) \vdash^W F : \beta}{C'''', (\Gamma, x : \forall\alpha.C'' \Rightarrow \alpha) \vdash^W F : \beta}}{C'''' \wedge \exists\alpha.C'', \Gamma \vdash^W (\mathbf{let}\ x = E\ \mathbf{in}\ F) : \beta}$$

  Von der Konsistenz wissen wir, daß $C \wedge \mathrm{ext}(\Gamma) \vdash^A \mathrm{ext}(\sigma)$ gilt. Außerdem haben wir die zwei Allgemeinheitsbedingungen der Induktionshypothese und die Kontextabschwächungsbedingung. Wir müssen nun

  $$\frac{\begin{array}{l} C \wedge \mathrm{ext}(\Gamma) \vdash^A \mathrm{ext}(\sigma) \quad C \wedge \mathrm{ext}(\sigma) \wedge \mathrm{ext}(\Gamma) \vdash^A \exists\alpha.C'' \wedge \alpha = \tau' \\ C' \wedge \mathrm{ext}(\Gamma, x : \sigma)) \vdash^A \exists\beta.C''' \wedge \beta = \tau \quad C''' \wedge C \wedge \mathrm{ext}(\Gamma) \vdash^A C'''' \end{array}}{C \wedge C' \wedge \mathrm{ext}(\Gamma) \vdash^A \exists\beta.((C'''' \wedge \exists\alpha.C'') \wedge \beta = \tau)}$$

  zeigen.

Aus der letzten Prämisse folgt:

$$\frac{\dfrac{C''' \wedge C \wedge \mathrm{ext}(\Gamma) \vdash^A C''''}{C''' \wedge C \wedge \mathrm{ext}(\Gamma) \wedge \beta = \tau \vdash^A C'''' \wedge \beta = \tau}}{C \wedge \mathrm{ext}(\Gamma) \wedge \exists\beta.C''' \wedge \beta = \tau \vdash^A \exists\beta.(C'''' \wedge \beta = \tau)}$$

Das können wir mit der dritten Prämisse zu

$$C \wedge C' \wedge \mathrm{ext}(\Gamma) \wedge \mathrm{ext}(\sigma) \vdash^A \exists\beta.(C'''' \wedge \beta = \tau)$$

kombinieren. Mit der zweiten Prämisse erhalten wir

$$\begin{array}{l} C \wedge C' \wedge \mathrm{ext}(\Gamma) \wedge \mathrm{ext}(\sigma) \\ \quad \vdash^A (\exists\alpha.C'') \wedge \exists\beta.(C'''' \wedge \beta = \tau) \ , \end{array}$$

und wegen der ersten Prämisse kann $\mathrm{ext}(\sigma)$ aus der Voraussetzung wegfallen. Durch Herausziehen von β erhalten wir dann die geforderte Behauptung.

- $$(fix)\frac{C, (\Gamma, x : \sigma) \vdash E : \sigma \quad C \vdash^A \mathrm{ext}(\sigma)}{C, \Gamma \vdash (\mathbf{fix}\ x :: P \Rightarrow \tau\ \mathbf{in}\ E) : \sigma} \left\{ \begin{array}{l} \bar{n}, \bar{\alpha} = \mathrm{fv}(P, \tau) \\ \sigma = \forall\bar{\alpha}.\forall\bar{n}.P \Rightarrow \tau \end{array} \right.$$

  Nach Induktion gibt es eine Ableitung

  $$\frac{C', (\Gamma, x : \forall\bar{\alpha}.\forall\bar{n}.P \Rightarrow \tau) \vdash^W E : \gamma}{\begin{array}{l} \exists\bar{\alpha}.\exists\bar{n}.(\delta = \tau \wedge P \wedge \forall\bar{\alpha}.\forall\bar{n}.(P \rightarrow \exists\alpha.(C \wedge \tau = \gamma))), \\ \quad \Gamma \vdash^W (\mathbf{fix}\ x :: P \Rightarrow \tau\ \mathbf{in}\ E) : \delta \end{array}}.$$

  Wir müssen jetzt noch

  $$\frac{C \wedge P \wedge \mathrm{ext}(\Gamma) \vdash^A \exists\gamma.C' \wedge \gamma = \tau}{\begin{array}{l} C \wedge P \wedge \mathrm{ext}(\Gamma) \\ \quad \vdash^A \exists\delta.(\exists\bar{\alpha}.\exists\bar{n}.(\delta = \tau \wedge P \wedge \forall\bar{\alpha}.\forall\bar{n}. \\ \quad (P \rightarrow \exists\gamma.(C' \wedge \tau = \gamma)))) \wedge \delta = \tau \end{array}}$$

  zeigen. Wir setzen dazu $\delta = \tau, \bar{\alpha} = \bar{\alpha}$ und $\bar{n} = \bar{n}$ und benennen die inneren Quantoren um: $\theta = [\bar{\beta}/\bar{\alpha}, \bar{m}/\bar{n}]$. Damit vereinfacht sich das Ziel zu

  $$\frac{C \wedge P \wedge \mathrm{ext}(\Gamma) \vdash^A \exists\gamma.C' \wedge \gamma = \tau}{C \wedge P \wedge \mathrm{ext}(\Gamma) \vdash^A P \wedge \forall\bar{\beta}.\forall\bar{m}.(\theta P \rightarrow \exists\gamma.(C' \wedge \theta\tau = \gamma))}.$$

P kann auf der rechten Seite weggelassen werden und $\bar{\beta}$ und $\bar{m}n$ tauchen links nicht auf, also kann auch diese Allquantifikation entfallen. Wir substituieren nun mit θ auf der Prämisse des Ziels und erhalten

$$C \wedge \theta P \wedge \mathrm{ext}(\Gamma) \vdash^A \exists\gamma.C' \wedge \gamma = \theta\tau.$$

Wir bringen θP auf die rechte Seite und haben unser Ziel erreicht.

- $$(\forall \Rightarrow I)\frac{C \wedge C', \Gamma \vdash E : \tau}{C \wedge \exists\bar{\alpha}.\exists\bar{n}.C', \Gamma \vdash E : \forall\bar{\alpha}.\forall\bar{n}.C' \Rightarrow \tau} \quad \bar{n}, \bar{\alpha} \notin \mathrm{fv}(C)$$

  Wir haben $C'', \Gamma \vdash^W E : \beta$ und müssen aus einer Allgemeinheitsbedingung die nächste zeigen.

  $$\frac{C \wedge C' \wedge \mathrm{ext}(\Gamma) \vdash^A \exists\beta.C'' \wedge \beta = \tau}{C \wedge (\exists\bar{\alpha}.\exists\bar{n}.C') \wedge C' \wedge \mathrm{ext}(\Gamma) \vdash^A \exists\beta.C'' \wedge \beta = \tau}$$

  gilt trivialerweise.

- $$(\forall \Rightarrow E)\frac{C, \Gamma \vdash E : \forall\bar{\alpha}.\forall\bar{n}.C' \Rightarrow \tau \quad C \vdash^A [\bar{\tau}/\bar{\alpha}][\bar{p}/\bar{n}]C'}{C, \Gamma \vdash E : [\bar{\tau}/\bar{\alpha}][\bar{p}/\bar{n}]\tau}$$

  Wir haben wieder $C'', \Gamma \vdash^W E : \beta$. Wir leiten mit

  $$\frac{\dfrac{C \wedge C' \wedge \mathrm{ext}(\Gamma) \vdash^A \exists\beta.C'' \wedge \beta = \tau'}{\begin{array}{c} C \wedge [\bar{\tau}/\bar{\alpha}][\bar{p}/\bar{n}]C' \wedge \mathrm{ext}(\Gamma) \\ \vdash^A \exists\beta.C'' \wedge \beta = [\bar{\tau}/\bar{\alpha}][\bar{p}/\bar{n}]\tau' \end{array}} \qquad C \vdash^A [\bar{\tau}/\bar{\alpha}][\bar{p}/\bar{n}]C'}{C \wedge \mathrm{ext}(\Gamma) \vdash^A \exists\beta.C'' \wedge \beta = [\bar{\tau}/\bar{\alpha}][\bar{p}/\bar{n}]\tau'}$$

  die neue Allgemeinheitsbedingung her.

- $$(\exists I)\frac{C, \Gamma \vdash E : \sigma}{\exists\bar{\alpha}.\exists\bar{n}.C, \Gamma \vdash E : \sigma} \qquad \bar{n}, \bar{\alpha} \notin \mathrm{fv}(\Gamma, \sigma)$$

Wir wissen, daß $C'', \Gamma \vdash^W E : \beta$ gilt. Mit $\sigma = \forall\bar{\gamma}.\forall\bar{m}.C_\sigma \Rightarrow \tau$ leiten wir

$$\frac{C \wedge C_\sigma \wedge \mathrm{ext}(\Gamma) \vdash^A \exists\beta.C'' \wedge \beta = \tau}{(\exists\bar{\alpha}.\exists\bar{n}.C) \wedge C_\sigma \wedge \mathrm{ext}(\Gamma) \vdash^A \exists\beta.C'' \wedge \beta = \tau}$$

her.

- $$(=)\frac{C, [\bar{\tau}/\bar{\alpha}][\bar{t}/\bar{n}]\Gamma \vdash E : [\bar{\tau}/\bar{\alpha}][\bar{t}/\bar{n}]\sigma \quad C \vdash^A \bar{t} = \bar{t}', \bar{\tau} = \bar{\tau}'}{C, [\bar{\tau}'/\bar{\alpha}][\bar{t}'/\bar{n}]\Gamma \vdash E : [\bar{\tau}'/\bar{\alpha}][\bar{t}'/\bar{n}]\sigma}$$

  Wir wissen, daß $C'', [\bar{\tau}/\bar{\alpha}][\bar{t}/\bar{n}]\Gamma \vdash^W E : \beta$ gilt. Aber wegen

  $$\frac{C \vdash^A \bar{t} = \bar{t}', \bar{\tau} = \bar{\tau}'}{C \vdash^A [\bar{\tau}/\bar{\alpha}][\bar{t}/\bar{n}]\Gamma \sqsubseteq [\bar{\tau}'/\bar{\alpha}][\bar{t}'/\bar{n}]\Gamma}$$

  bekommen wir mit der Kontextabschwächung

  $$C''', [\bar{\tau}/\bar{\alpha}][\bar{t}/\bar{n}]\Gamma \vdash^W E : \beta.$$

  Die verbleibende Bedingung

  $$\frac{\begin{array}{l} C'' \wedge C \vdash^A C''' \quad C \vdash^A \bar{t} = \bar{t}', \bar{\tau} = \bar{\tau}' \\ C \wedge [\bar{\tau}/\bar{\alpha}][\bar{t}/\bar{n}]P \wedge \mathrm{ext}([\bar{\tau}/\bar{\alpha}][\bar{t}/\bar{n}]\Gamma) \\ \quad \vdash^A \exists\beta.C'' \wedge \beta = [\bar{\tau}/\bar{\alpha}][\bar{t}/\bar{n}]\tau' \end{array}}{\begin{array}{l} C \wedge [\bar{\tau}'/\bar{\alpha}][\bar{t}'/\bar{n}]P \wedge \mathrm{ext}([\bar{\tau}'/\bar{\alpha}][\bar{t}'/\bar{n}]\Gamma) \\ \quad \vdash^A \exists\beta.C''' \wedge \beta = [\bar{\tau}'/\bar{\alpha}][\bar{t}'/\bar{n}]\tau' \end{array}}$$

  gilt.

- $$(\textbf{data } I)\frac{\texttt{data T}\bar{\alpha}\bar{n} = \sum_i D_i\bar{\tau}_i \Leftarrow Q_i, \bar{\beta}_i\bar{m}_i \quad C \vdash^A \exists\bar{m}_i.\exists\bar{n}.Q_i}{C, \Gamma \vdash D_i : \forall\bar{m}_i.\forall\bar{n}.\forall\bar{\beta}_i.\forall\bar{\alpha}.Q_i \Rightarrow (\bar{\tau}_i \to T\bar{\alpha}\bar{n})}$$

  Wir haben eine Ableitung

  $$\frac{\texttt{data } T\bar{\alpha}\bar{n} = \sum_i D_i\bar{\tau}_i \Leftarrow Q_i, \bar{\beta}_i\bar{m}_i}{\exists\bar{\alpha}.\exists\bar{\beta}_i.\exists\bar{n}.\exists\bar{m}_i.(\delta = \bar{\tau}_i \to T\bar{\alpha}\bar{n} \wedge Q_i), \Gamma \vdash^W D_i : \delta}$$

  Jetzt müssen wir noch

  $$\begin{array}{l} C \wedge Q_i \wedge \mathrm{ext}(\Gamma) \vdash^A \exists\delta.(\exists\bar{\alpha}.\exists\bar{\beta}_i.\exists\bar{n}.\exists\bar{m}_i. \\ \quad (\delta = \bar{\tau}_i \to T\bar{\alpha}\bar{n} \wedge Q_i)) \wedge (\delta = \bar{\tau}_i \to T\bar{\alpha}\bar{n}) \end{array}$$

  überprüfen. Wir setzen einfach $\delta = \bar{\tau}_i \to T\bar{\alpha}\bar{n}$, $\bar{\alpha} = \bar{\alpha}$, $\bar{\beta}_i = \bar{\beta}_i$, $\bar{n} = \bar{n}$ und $\bar{m}_i = \bar{m}_i$.

- $$(\textbf{data } E)\frac{\begin{array}{l}\texttt{data T}\bar\alpha\bar n = \sum_i D_i\bar\tau_i \Leftarrow Q_i, \bar\beta_i\bar m_i\\ (C \wedge \theta Q_i), (\Gamma, \bar y_i : \theta\bar\tau_i) \vdash E_i : \tau\\ C, \Gamma \vdash F : \theta(T\bar\alpha\bar n)\end{array}}{C, \Gamma \vdash \textbf{case } F \textbf{ of } \{D_i\bar y_i \Rightarrow E_i\} : \tau}\left\{\begin{array}{c}\theta = [\bar m_i'/\bar m_i, \bar n'/\bar n,\\ \bar\mu/\bar\alpha, \bar\nu_i/\bar\beta_i]\\ \bar m_i', \bar\nu_i \notin \mathrm{fv}(C, \Gamma, \tau)\end{array}\right.$$

  Per Induktion erhalten wir $\vdash^W$-Ableitungen $C_i', (\Gamma, \bar y_i : \bar\theta\bar\tau_i) \vdash^W E_i : \zeta_i$ und $C', \Gamma \vdash^W F : \eta$. Wir schließen

  $$\theta\bar\tau_i = \bar\rho_i \vdash^A (\Gamma, \bar y_i : \theta\bar\tau_i) \sqsubseteq (\Gamma, \bar y_i : \bar\rho_i),$$

  um nach Kontextabschwächung die $\vdash^W$ Regel

  $$\frac{\begin{array}{l}\texttt{data } T\bar\alpha\bar n = \sum_i D_i\bar\tau_i \Leftarrow Q_i, \bar\beta_i\bar m_i\\ C', \Gamma \vdash^W F : \eta\\ C_i'', (\Gamma, \bar y_i : \bar\rho_i) \vdash^W E_i : \zeta_i\end{array}}{\begin{array}{l}\exists\eta.\exists\bar\alpha.\exists\bar n.(\eta = T\bar\alpha\bar n \wedge C' \wedge\\ \quad \forall i.\forall\bar\beta_i\forall\bar m_i.\exists\zeta_i.\exists\bar\rho_i.(Q_i \to (\bar\rho_i = \bar\tau_i \wedge C_i'' \wedge \zeta_i = \delta))),\\ \quad \Gamma \vdash^W \textbf{case } F \textbf{ of } \{D_i\bar y_i \Rightarrow E_i\} : \delta\end{array}}$$

  anwenden zu können. Wieder verbleibt die Überprüfung einer Bedingung.

  $$\frac{\begin{array}{l}\bar\rho_i = \theta\bar\tau_i \wedge C_i' \vdash^A C_i'' \quad C \wedge \mathrm{ext}(\Gamma) \vdash^A \exists\eta.C' \wedge \eta = \theta(T\bar\alpha\bar n)\\ \quad C \wedge \theta Q_i \wedge \mathrm{ext}(\Gamma, \bar y_i : \theta\bar\tau_i) \vdash^A \exists\zeta_i.C_i' \wedge \zeta_i = \tau\end{array}}{\begin{array}{l}C \wedge \mathrm{ext}(\Gamma) \vdash^A \exists\delta.(\exists\eta.\exists\bar\alpha.\exists\bar n.(\eta = T\bar\alpha\bar n \wedge C' \wedge\\ \quad \forall i.\forall\bar\beta_i.\forall\bar m_i.\exists\zeta_i.\exists\bar\rho_i.(Q_i \to (\bar\rho_i = \bar\tau_i \wedge C_i'' \wedge \zeta_i = \delta)))) \wedge \delta = \tau\end{array}}$$

  Wir können die Behauptung vereinfachen, indem wir $\delta = \tau, \bar\alpha = \theta\bar\alpha = \bar\mu$ und $\bar n = \theta\bar n = \bar n'$ wählen. Es entsteht

  $$C \wedge \mathrm{ext}(\Gamma) \vdash^A (\exists\eta.\eta = \theta(T\bar\alpha\bar n) \wedge C') \wedge \forall i.\forall\bar\beta_i\forall\bar m_i.\exists\zeta_i.\exists\bar\rho_i.(\dots)$$

  Der erste Teil der verbliebenen zweiten Behauptung ist gerade eine der Voraussetzungen. Um die restliche allquantifizierte Aussage zu zeigen, zeigen wir sie für $\theta\bar\beta_i = \bar\nu_i, \theta\bar m_i = \bar m_i'$ allgemein:

Wir folgern aus der ersten Voraussetzung

$$\dfrac{\dfrac{\dfrac{\bar{\rho}_i = \theta\bar{\tau}_i \wedge C_i' \vdash^A C_i''}{\bar{\rho}_i = \theta\bar{\tau}_i \wedge C_i' \vdash^A \bar{\rho}_i = \theta\bar{\tau}_i \wedge C_i''}}{\exists\bar{\rho}_i.(\bar{\rho}_i = \theta\bar{\tau}_i) \wedge C_i' \vdash^A \exists\bar{\rho}_i.(\bar{\rho}_i = \theta\bar{\tau}_i \wedge C_i'')}}{C_i' \vdash^A \exists\bar{\rho}_i.(\bar{\rho}_i = \theta\bar{\tau}_i \wedge C_i'')} \quad ,$$

bringen in der letzten Voraussetzung θQ_i auf die andere Seite und benutzen $\mathrm{ext}(\theta\bar{\tau}_i) = \mathrm{true}$. Es bleibt nur

$$\dfrac{\begin{array}{c} C_i' \vdash^A \exists\bar{\rho}_i.(\bar{\rho}_i = \theta\bar{\tau}_i \wedge C_i'') \\ C \wedge \mathrm{ext}(\Gamma) \vdash^A \theta Q_i \rightarrow \exists\zeta_i.C_i' \wedge \zeta_i = \tau \end{array}}{C \wedge \mathrm{ext}(\Gamma) \vdash^A \exists\zeta_i.\exists\bar{\rho}_i.(\theta Q_i \rightarrow (\bar{\rho}_i = \theta\bar{\tau}_i \wedge \theta C_i'' \wedge \zeta_i = \theta\tau))}$$

zu zeigen. Dies tun wir, indem wir die erste Voraussetzung in der zweiten einsetzen und die Quantoren nach außen ziehen.

□

A.3 Unifikation

Ziel dieses Abschnitts ist es, die Korrektheit und Vollständigkeit der Unifikation zu zeigen. Die Unifikation besteht aus einigen Algorithmen (**R**, **M**, $\mathcal{T}_1$, $\mathcal{T}_2$), die auf Typconstraints arbeiten. Wir werden jeweils Aussagen über Form (syntaktische Eigenschaften des Resultats) und Korrektheit (semantische Eigenschaften des Resultats) machen. Doch bevor wir dazu kommen, wollen wir das newinst-Lemma zeigen.

A.3.1 Das newinst-Lemma

Die folgenden drei Aussagen gehören zusammen. Ziel ist es zu zeigen, daß mit $C^\bullet \vdash \psi$ und $(\bar{\beta}, \bar{n}, \psi') = \text{newinst}(\psi)$ auch $C \vdash \exists \bar{\beta}, \bar{n}.\psi'$ gilt.

Hilfssatz 7. *Wenn $\mu^\bullet = [\![\tau^\bullet]\!]_{\rho'}$ für irgendein ρ' und $(\bar{\beta}, \bar{n}, \tau') = \text{newinst}(\tau)$ gelten, dann gilt für alle ρ:*

$$\exists \bar{\beta}', \bar{n}'.(\mu = [\![\tau']\!]_{\rho[\bar{\beta}'/\bar{\beta}, \bar{n}'/\bar{n}]})$$

Beweis:

Wir führen eine Induktion über die Struktur von τ.

- $\tau = \gamma$:
  Also gilt auch $\tau^\bullet = \gamma$. Aus der Definition von newinst wissen wir, daß $\tau' = \beta$ für eine neue Variable β. Wir müssen nun $\exists \beta'.\mu = [\![\beta]\!]_{\rho[\beta'/\beta]}$ zeigen. Aber mit $\beta' = \mu$ ist die Gleichung offensichtlich erfüllt und damit die Existenz eines β' gesichert.

- $\tau = \tau_1 \to \tau_2$:
  Also gilt auch $\tau' = \tau_1' \to \tau_2'$, $(\bar{\beta}_1, \bar{n}_1, \tau_1') = \text{newinst}(\tau_1)$ und $(\bar{\beta}_2, \bar{n}_2, \tau_2') = \text{newinst}(\tau_2)$. Weiter gilt

  $$\mu^\bullet = [\![\tau^\bullet]\!]_{\rho'} = [\![\tau_1^\bullet \to \tau_2^\bullet]\!]_{\rho'} = [\![\tau_1^\bullet]\!]_{\rho'} \to [\![\tau_2^\bullet]\!]_{\rho'},$$

  also auch $\mu = \mu_1 \to \mu_2$ und mit der Induktionshypothese

  $$\exists \bar{\beta}_1', \bar{n}_1'.(\mu_1 = [\![\tau_1']\!]_{\rho[\bar{\beta}_1'/\bar{\beta}_1, \bar{n}_1'/\bar{n}_1]})$$
  $$\exists \bar{\beta}_2', \bar{n}_2'.(\mu_2 = [\![\tau_2']\!]_{\rho[\bar{\beta}_2'/\bar{\beta}_2, \bar{n}_2'/\bar{n}_2]}).$$

Wir können die $\bar{\beta}_i$ und $\bar{n}_i$ ausklammern und erhalten das gewünschte Ergebnis:

$$\exists \bar{\beta}'_1, \bar{\beta}'_2, \bar{n}'_1, \bar{n}'_2.(\mu = [\![\tau']\!]_{\rho[\bar{\beta}'_1/\bar{\beta}_1, \bar{\beta}'_2/\bar{\beta}_2, \bar{n}'_1/\bar{n}_1, \bar{n}'_2/\bar{n}_2]})$$

- $\tau = \mathtt{T}\bar{\tau}\bar{t}$:

  Nun ist auch $\tau' = \mathtt{T}\bar{\tau}'\bar{m}$ mit $(\bar{\beta}_i, \bar{n}_i, \tau'_i) = \text{newinst}(\tau_i)$, und $\bar{m}$ neue Variablen. Wir haben

  $$\mu^\bullet = [\![\tau^\bullet]\!]_{\rho'} = [\![\mathtt{T}\bar{\tau}^\bullet \bar{c}_s]\!]_{\rho'} = \mathtt{T}([\![\bar{\tau}^\bullet]\!]_{\rho'}, \bar{A}_{c_s}).$$

  Also gilt auch $\mu = \mathtt{T}\bar{\nu}\bar{r}$,

  $$\exists \bar{\beta}'_i, \bar{n}'_i.(\nu_i = [\![\tau'_i]\!]_{\rho[\bar{\beta}'_i/\bar{\beta}_i, \bar{n}'_i/\bar{n}_i]})$$

  und natürlich

  $$\exists \bar{m}'.\bar{m}' = \bar{r}.$$

  Nach Herausziehen der Quantoren gilt

  $$\exists \bar{\bar{\beta}}', \bar{\bar{n}}', \bar{m}'.(\mu = [\![\tau']\!]_{\rho[\bar{\bar{\beta}}'/\bar{\bar{\beta}}, \bar{\bar{n}}'/\bar{\bar{n}}, \bar{m}'/\bar{m}]}).$$

□

Hilfssatz 8. *Wenn* $C^\bullet \vdash^A \alpha = \tau$ *und* $(\bar{\beta}, \bar{n}, \tau') = \text{newinst}(\tau)$ *gelten, dann gilt auch*

$$C \vdash^A \exists \bar{\beta}, \bar{n}.\alpha = \tau'.$$

Beweis:
Wir müssen

$$\pi_i([\![C]\!]_\rho) \to \pi_i([\![\exists \bar{\beta}, \bar{n}.\alpha = \tau']\!]_\rho)$$

für $i = 1$ und $i = 2$ zeigen. Da $\pi_1([\![C]\!]_\rho) \to \pi_2([\![C]\!]_\rho)$ gilt, reicht es,

$$\pi_2([\![C]\!]_\rho) \to \pi_1([\![\exists \bar{\beta}, \bar{n}.\alpha = \tau']\!]_\rho)$$

zu zeigen.

Nehmen wir also $\pi_2([\![C]\!]_\rho)$ an. Das ist das Gleiche wie $\pi_2([\![C^\bullet]\!]_{\rho_\bullet})$. Mit der Voraussetzung haben wir nun $\pi_2([\![\alpha = \tau]\!]_{\rho_\bullet})$. Dies wiederum impliziert

$$\rho(\alpha)^\bullet \cong [\![\tau]\!]_{\rho_\bullet}$$

Aus einem weiteren Hilfssatz am Anfang wissen wir, daß $[\![\tau]\!]_{\rho_\bullet} \cong [\![\tau^\bullet]\!]_{\rho_\bullet}$ ist und somit

$$\rho(\alpha)^\bullet \cong [\![\tau^\bullet]\!]_{\rho_\bullet}$$

gilt. Mit dem Hilfssatz von oben ergibt sich:

$$(\exists \bar\beta', \bar n'.(\rho(\alpha) = [\![\tau']\!]_{\rho[\bar\beta'/\bar\beta, \bar n'/\bar n]})) = \pi_1([\![\exists \bar\beta, \bar n.\alpha = \tau']\!]_\rho)$$

Damit ist die Behauptung bewiesen. □

Lemma 3 (newinst)**.** *Wenn* $C^\bullet \vdash^A \psi$ *und* $(\bar\beta, \bar n, \psi') = \text{newinst}(\psi)$ *gelten, dann gilt auch* $C \vdash^A \exists \bar\beta, \bar n.\psi'$.

Beweis:
Das Lemma folgt einfach aus dem letzten Hilfssatz: Wenn $\psi = [\bar\tau/\bar\alpha]$ ist, dann gilt ja $C^\bullet \vdash^A \alpha_i = \tau_i$ für alle i. Also gilt $C \vdash^A \exists \bar\beta_i.\exists \bar n_i.\alpha_i = \tau_i$, und nach Ausklammern folgt die Behauptung. □

Hilfssatz 9. *Wenn* $(\bar\beta, \bar n, \tau') = \text{newinst}(\tau)$ *ist, dann ist*

$$(\tau' = \tau)^\bullet \equiv^A \bar\gamma = \bar\gamma'$$

für irgendwelche Variablen γ *und* γ'.

Beweis:
Der Beweis ist eine einfache Induktion über die Definition von newinst.

- $\tau = \alpha$
  Es gilt

  $$(\tau' = \tau)^\bullet = (\beta = \alpha)^\bullet = (\beta = \alpha).$$

- $\tau = \tau_1 \to \tau_2$
  Es gilt

  $$\begin{aligned}(\tau_1' \to \tau_2' = \tau_1 \to \tau_2)^\bullet &= (\tau_1'^\bullet \to \tau_2'^\bullet = \tau_1^\bullet \to \tau_2^\bullet) \\ &\equiv^A (\tau_1'^\bullet = \tau_1^\bullet) \wedge (\tau_2'^\bullet = \tau_2^\bullet) = (\tau_1' = \tau_1)^\bullet \wedge (\tau_2' = \tau_2)^\bullet\end{aligned},$$

  was nach Induktionsbehauptung die geforderte Form hat.

- $\tau = T\bar{\tau}\bar{n}$
  Es gilt

$$(T\bar{\tau}'\bar{n} = T\bar{\tau}\bar{t})^\bullet = (T\bar{\tau'}^\bullet\bar{c}_s = T\bar{\tau}^\bullet\bar{c}_s)$$
$$\equiv^A \bigwedge_i(\tau_i'^\bullet = \tau_i^\bullet) = \bigwedge_i(\tau_i' = \tau_i)^\bullet \ ,$$

  was wieder die geforderte Form hat.

A.3.2 Form und Korrektheit von R

Die folgende Proposition besagt, daß nach Anwendung des Algorithmus **R** auf eine strukturell erfüllte Gleichung, die zweite Komponente des Resultats nur aus Variablengleichungen mit Bedingung besteht. Dazu zeigen wir erst einen Hilfssatz.

Hilfssatz 10. *Wenn* $\overline{\alpha_i = \alpha_j} \vdash \tau^\bullet = \tau'^\bullet$ *gilt, dann gilt auch* $\mathbf{R}(\tau = \tau') = (P, \overline{\gamma_i = \gamma_j})$.

Beweis:
Wir zeigen das per Induktion über die Struktur des Arguments von **R**, $\tau = \tau'$:

- $\tau_1 \to \tau_2 = \tau_1' \to \tau_2'$

$$[\![\tau^\bullet = \tau'^\bullet]\!]_\rho = [\![\tau_1^\bullet = \tau_1'^\bullet]\!]_\rho \wedge [\![\tau_2^\bullet = \tau_2'^\bullet]\!]_\rho$$

  Wir können also die Induktionshypothese anwenden und die Behauptung folgt trivial.

- $\mathtt{T}\bar{\tau}\bar{t} = \mathtt{T}\bar{\tau}'\bar{t}'$

  Auch hier können wir die Induktionshypothese auf $\mathbf{R}(\tau_i = \tau_i')$ anwenden. Also ist $\mathbf{R}(\bar{\tau} = \bar{\tau}') = (P, \overline{\gamma_i = \gamma_j})$ und $\mathbf{R}(\tau = \tau') = (P \wedge \bar{t} = \bar{t}', \overline{\gamma_i = \gamma_j})$.

- $\alpha = \tau$

  Nehmen wir zunächst einmal an, τ sei nicht eine einzelne Variable. Es läßt sich dann ein Typ τ' finden, so daß für alle ρ $[\![\tau^\bullet]\!]_\rho \neq \tau'$. Wir setzen nun $\rho = [\tau'/\bar{\alpha}]$. Nach Voraussetzung gilt $\overline{\alpha_i = \alpha_j} \vdash \alpha^\bullet = \tau^\bullet$ also auch

$$\pi_1([\![\overline{\alpha_i = \alpha_j}]\!]_\rho) \to \pi_1([\![\alpha = \tau^\bullet]\!]_\rho)$$

Die linke Seite ist wahr, die rechte aber nicht, wir haben einen Widerspruch.

τ ist also eine einzelne Variable β und $\mathbf{R}(\alpha = \beta) = (\text{true}, \alpha = \beta)$, was von der geforderten Form ist.

□

Proposition 14 (Form und Korrektheit von R). *Wenn C ein quantorenfreier Constraint ist, dann ist $\mathbf{R}(C) = (P, G)$ und es gilt $C \equiv^A P \wedge G$, wobei P ein Termconstraint ist und G von der Form $\bigwedge_i (P_i \to \alpha_i = \tau_i)$ ist.*

Wenn weiter $C = \psi' C'$, $\vdash^A \psi C'^\bullet$ und $(\psi', \bar{\beta}, \bar{m}) = \text{newinst}(\psi)$ gilt, dann haben die Typen τ_i die Form einer einzelnen Variablen β_i.

Beweis:
Der Beweis der Korrektheit ist ein triviale strukturelle Induktion über die Struktur von C.

Wir beweisen die Formaussage mit einer strukturellen Induktion über die Struktur von C':

- $C_1 \wedge C_2$

  trivial.

- $\text{inj}(P')$

  trivial.

- $\tau = \tau'$

  Wir wissen $\vdash^A \psi\tau^\bullet = \psi\tau'^\bullet$. Also gilt $\vdash^A (\psi\tau^\bullet)^\bullet = (\psi\tau'^\bullet)^\bullet$, was das gleiche ist wie $\vdash^A (\psi^\bullet\tau^\bullet) = (\psi^\bullet\tau'^\bullet)$. Es folgt

  $$\psi'^\bullet \wedge \psi^\bullet \vdash^A \psi'^\bullet\tau^\bullet = \tau^\bullet = \psi^\bullet\tau^\bullet = \psi^\bullet\tau'^\bullet = \tau'^\bullet = \psi'^\bullet\tau'^\bullet$$

  und nach Substitution mit $\psi'^\bullet$

  $$\psi'^\bullet\psi^\bullet \vdash^A \psi'^\bullet\tau^\bullet = \psi'^\bullet\tau'^\bullet.$$

  $\psi'^\bullet\psi^\bullet$ ist von der Form $\bar{\tau}'^\bullet = \bar{\tau}^\bullet$. Da jedes τ'_i aus τ durch newinst entstanden ist, gilt

  $$\bar{\tau}'^\bullet = \bar{\tau}^\bullet \equiv^A \overline{\gamma_i = \gamma_j},$$

und damit haben wir

$$\overline{\gamma_i = \gamma_j} \vdash^A (\psi'\psi)^\bullet \vdash (\psi'\tau)^\bullet = (\psi'\tau')^\bullet.$$

Nun können wir mit dem Hilfssatz von oben behaupten:

$$\mathbf{R}(\psi'\tau = \psi'\tau') = (P, G).$$

- $P' \to C'$

  Wir haben $\vdash^A \psi C'^\bullet$ und folglich $\mathbf{R}(\psi' C') = (P'', G'')$ und $\mathbf{R}(\psi' C) = ((P' \to P''), (P' \to G''))$.

□

A.3.3 Form und Korrektheit von M

Proposition 15 (Form und Korrektheit M). *Wenn* $\mathbf{M}(U, G) = (\psi', P, G', \bar{\beta}, \bar{n})$ *ist, dann gilt für alle* P' *mit* $\bar{\beta}, \bar{n} \notin \mathrm{fv}(P')$, *daß*

$$\exists\bar{\beta}.\exists\bar{n}.(P' \to P) \wedge \psi' \wedge (P' \to G') \equiv^A P' \to (\psi \wedge G),$$

wobei $\psi' G' = G'$ *und* G' *von der Form* $\bigwedge_i P_i \to (\alpha_i = \beta_i)$ *ist. Wenn* **M** *fehlschlägt, ist* $P' \to (U \wedge G) \equiv^A$ false.

Beweis:

$U^\bullet \wedge G^\bullet$ ist ein Unifikationsproblem im Sinne einer Standardunifikation. Da ψ'' die allgemeinste Lösung dieses Unifikationsproblems ist, gilt $\vdash^A \psi''(U^\bullet \wedge G^\bullet)$ und nach der Anwendung von **R**bekommen wir $\psi'(U \wedge G) \equiv^A P \wedge G'$.

Wir beachten, daß $(P \to (U \wedge G))^\bullet \equiv^A U^\bullet \wedge G^\bullet$ gilt und mit dem newinst-Lemma haben wir:

$$P' \to (U \wedge G) \vdash^A \exists\bar{\beta}.\exists\bar{n}.\psi'$$

Also haben wir insgesamt:

$$\begin{aligned}
&P' \to (U \wedge G)\\
&\equiv^A (\exists\bar{\beta}.\exists\bar{n}.\psi') \wedge P' \to (U \wedge G)\\
&\quad\equiv^A \exists\bar{\beta}.\exists\bar{n}.(\psi' \wedge \psi'(P' \to (U \wedge G))\\
&\quad\equiv^A \exists\bar{\beta}.\exists\bar{n}.\psi' \wedge (P' \to (P \wedge G'))\\
&\quad\equiv^A \exists\bar{\beta}.\exists\bar{n}.\psi' \wedge ((P' \to P) \wedge (P' \to G'))
\end{aligned}$$

□

A.3.4 Form und Korrektheit von $\mathcal{T}_1$

Proposition 16 (Form und Korrektheit $\mathcal{T}_1$). *Es gilt $\mathcal{T}_1(C) \equiv^A C$. Außerdem ist daß $\mathcal{T}_1(C)$ beinahe in Normalform.*

Beweis:

Wir zeigen die Behauptung über Regel-Induktion und beschränken uns auf nichttriviale Fälle.

- $$\frac{\begin{array}{l}\mathcal{T}_1(C) = Q.P \wedge \psi \wedge G \quad \mathcal{T}_1(C') = Q'.P' \wedge \psi' \wedge G' \\ (\psi'', P'', G'', \bar{\beta}, \bar{n}) = \mathbf{M}(\psi \wedge \psi', G \wedge G')\end{array}}{\mathcal{T}_1(C \wedge C') = Q.Q'.\exists\bar{\beta}.\exists\bar{n}.(P \wedge P' \wedge P'') \wedge \psi'' \wedge G''}$$

  Aus der Korrektheit von $\mathbf{M}$ wissen wir, daß

  $$\psi \wedge \psi' \wedge G \wedge G' \equiv^A \exists\bar{\beta}.\exists\bar{n}.\psi'' \wedge P'' \wedge G''$$

  gilt. Also gilt auch

  $$\begin{array}{l} C \wedge C' \equiv^A \mathcal{T}_1(C) \wedge \mathcal{T}_1(C') \equiv^A Q.Q'.P \wedge P' \wedge \psi \wedge \psi' \wedge G \wedge G' \\ \equiv^A Q.Q'.P \wedge P' \wedge \exists\bar{\beta}.\exists\bar{n}.\psi'' \wedge P'' \wedge G'' \equiv^A \mathcal{T}_1(C \wedge C') \end{array} .$$

- $$\frac{\mathcal{T}_1(C) = Q.P \wedge \psi \wedge G \quad (\psi', P'', G'', \bar{\beta}, \bar{n}) = \mathbf{M}(\psi, \text{true})}{\mathcal{T}_1(P' \to C) = Q.\exists\bar{\beta}.\exists\bar{n}.(P' \to (P \wedge P'')) \wedge \psi' \wedge (P' \to (G \wedge G''))}$$

  Wieder schließen wir mit der Korrektheit von $\mathbf{M}$, daß

  $$P' \to \psi \equiv^A \exists\bar{\beta}.\exists\bar{n}.\psi' \wedge P' \to P'' \wedge P' \to G''$$

  gilt. Es folgt

  $$\begin{array}{l} P' \to C \equiv^A P' \to \mathcal{T}_1(C) \\ \equiv^A Q.P' \to P \wedge (P' \to \psi) \wedge (P' \to G) \\ \equiv^A Q.P' \to P \wedge (\exists\bar{\beta}.\exists\bar{n}.\psi' \wedge P' \to P'' \\ \qquad \wedge (P' \to G'')) \wedge (P' \to G) \\ \equiv^A \mathcal{T}_1(P' \to C) \end{array} .$$

□

A.3.5 Form und Korrektheit von $\mathcal{T}_2$

Proposition 17 (Form und Korrektheit $\mathcal{T}_2$). *Es gilt $\mathcal{T}_2(C) \equiv^A C$. Wenn C beinahe in Normalform war, dann ist $\mathcal{T}_2(C)$ in Normalform.*

Beweis:

Wir beweisen das wieder mit einer strukturellen Induktion über die Regeln und betrachten dabei wieder nur nichttriviale Regeln.

- $$\frac{\begin{array}{c}\mathcal{T}_2(C) = Q.P \wedge (\psi \wedge \alpha = \tau) \wedge G \\ (\psi', P', G', \bar{\beta}, \bar{n}) = \mathbf{M}([\tau/\alpha], \emptyset)\end{array}}{\mathcal{T}_2(\exists\alpha.C) = \exists\bar{\beta}.\exists\bar{n}.Q.(P \wedge P') \wedge \psi \wedge (G \wedge G')} \quad \alpha \notin \mathrm{dom}(\psi)$$

  Wir wissen aus der Korrektheit von $\mathbf{M}$, daß

  $$\alpha = \tau \equiv^A \exists\bar{\beta}.\exists\bar{n}.P' \wedge \psi' \wedge G'$$

  gilt. Damit haben wir erstens

  $$\begin{array}{l} Q.(P \wedge \psi \wedge \alpha = \tau \wedge G) \\ \quad \vdash^A Q.\alpha = \tau \vdash^A \exists\,\mathrm{fv}(\tau).\alpha = \tau \\ \quad \vdash^A \exists\bar{\beta}.\exists\bar{n}.\psi' \end{array}$$

  und zweitens nach Substitution mit ψ'

  $$\psi'\alpha = \tau \equiv^A P' \wedge G'.$$

  Insgesamt ergibt sich:

  $$\begin{array}{l} \exists\alpha.Q.(P \wedge \psi \wedge \alpha = \tau \wedge G) \\ \quad \equiv^A \exists\alpha.(\exists\bar{\beta}.\exists\bar{n}.\psi') \wedge Q.(P \wedge \psi \wedge \alpha = \tau \wedge G) \\ \quad \equiv^A \exists\alpha.\exists\bar{\beta}.\exists\bar{n}.\psi' \wedge Q.(P \wedge \psi \wedge \alpha = \tau \wedge G) \\ \quad \equiv^A \exists\bar{\beta}.\exists\bar{n}.\exists\alpha.\psi' \wedge Q.(P \wedge \psi \wedge \psi'\alpha = \tau \wedge G) \\ \quad \equiv^A \exists\bar{\beta}.\exists\bar{n}.(\exists\alpha.\psi') \wedge Q.(P \wedge \psi \wedge (P' \wedge G') \wedge G) \\ \quad \equiv^A \exists\bar{\beta}.\exists\bar{n}.Q.(P \wedge P') \wedge \psi \wedge (G \wedge G'). \end{array}$$

- $$\frac{\mathcal{T}_2(C) = Q.P \wedge \psi \wedge G}{\mathcal{T}_2(\forall\alpha.C) = \text{false}} \quad \alpha R_G^* \alpha', \alpha' \in \text{fv}(\psi, \forall\alpha.C)$$

  Als erstes gilt:

  $$\alpha R_G \alpha' \wedge \pi_2([\![G]\!]_\rho) \rightarrow \rho(\alpha) = \rho(\alpha')$$

  Zweitens gilt mit

  $$\pi_2([\![Q.C \wedge C']\!]_\rho)$$

  auch

  $$\exists\rho'.\pi_2([\![C]\!]_{\rho'}) \wedge \forall\beta \in \text{fv}(Q.C \wedge C').\rho(\beta) = \rho'(\beta).$$

  Nun nehmen wir $\pi_2([\![\forall\alpha.C]\!]_\rho)$ an. Wir unterscheiden zwei Fälle:

  - $\alpha' \in \text{fv}(\forall\alpha.C)$
    Es gilt

    $$\begin{array}{l}\pi_2([\![\forall\alpha.C]\!]_\rho) \\ \rightarrow \pi_2([\![G]\!]_{\rho'[\texttt{int}/\alpha]}) \wedge \pi_2([\![G]\!]_{\rho'[\texttt{bool}/\alpha]}) \\ \rightarrow \rho'(\alpha') \cong \texttt{int} \wedge \rho'(\alpha') \cong \texttt{bool}\end{array}$$

    und wir haben den Widerspruch.
  - $\alpha' \in \text{fv}(\psi)$, sei $\alpha' \in \text{fv}(\tau)$ und $[\tau/\gamma]$ in ψ.
    Es gilt:

    $$\begin{array}{l}\pi_2([\![\forall\alpha.C]\!]_\rho) \\ \rightarrow \pi_2([\![C]\!]_{\rho[\texttt{int}/\alpha]}) \wedge \pi_2([\![C]\!]_{\rho[\texttt{bool}/\alpha]}) \\ \quad \rightarrow \pi_2([\![\gamma = \tau]\!]_{\rho'}) \wedge \rho'(\alpha') = \texttt{int} \\ \quad \wedge\pi_2([\![\gamma = \tau]\!]_{\rho''}) \wedge \rho''(\alpha') = \texttt{bool} \\ \rightarrow \rho'(\gamma) \cong \rho'(\tau) \wedge \rho'(\alpha') = \texttt{int} \\ \quad \wedge\rho''(\gamma) \cong \rho''(\tau) \wedge \rho''(\alpha') = \texttt{bool}\end{array}$$

    was zu einem Widerspruch führt, da

    $$\rho'(\gamma) = \rho(\gamma) = \rho''(\gamma)$$

    gilt, aber $\alpha' \in \text{fv}(\tau)$ ist.

Jetzt gilt also für alle ρ, daß $\pi_2([\![\forall\alpha.C]\!]_\rho)$ falsch ist und somit auch $\pi_1([\![\forall\alpha.C]\!]_\rho)$ und damit $[\![\forall\alpha.C]\!]_\rho = (\text{false}, \text{false})$.

- $$\frac{\mathcal{T}_2(C) = Q.P \wedge \psi \wedge (G' \wedge G'')}{\mathcal{T}_2(\forall\alpha.C) = Q.P \wedge \psi \wedge G''} \quad \begin{cases} \text{fv}(G') = \{\alpha' \mid \alpha R_G^* \alpha'\} \\ \text{fv}(G'', \psi, \forall\alpha.C) \cap \text{fv}(G') = \emptyset \end{cases}$$

  Sei $\bar{\beta} = \text{fv}(G')$ und Q_β gleich Q mit allen $\bar{\beta}$ entfernt. Es gilt offensichtlich

  $$\forall\alpha.Q.P \wedge \psi \wedge G' \wedge G'' \vdash^A \forall\alpha.Q.P \wedge \psi \wedge G'' \vdash^A Q.P \wedge \psi \wedge G''$$

  Die andere Richtung ist schwieriger. Sei t beliebig aber fest. Es gilt

  $$\begin{array}{l} Q.P \wedge \psi \wedge G'' \\ \equiv^A Q_\beta.P \wedge \psi \wedge G'' \\ \vdash^A [t/\alpha][t/\bar{\beta}]Q_\beta.P \wedge \psi \wedge G' \wedge G'' \ , \\ \vdash^A [t/\alpha]\exists\bar{\beta}.Q_\beta.P \wedge \psi \wedge G' \wedge G'' \\ \vdash^A [t/\alpha]Q.P \wedge \psi \wedge G' \wedge G'' \end{array}$$

  und weil t allgemein war gilt auch

  $$Q.P \wedge \psi \wedge G'' \vdash^A \forall\alpha.Q.P \wedge \psi \wedge G' \wedge G''.$$

Es bleibt zu bemerken, daß immer genau eine der beide $(\forall\alpha.C)$-Regeln anwendbar ist. Wenn nämlich

$$\{\alpha' \mid \alpha R_G^* \alpha'\} \cap \text{fv}(\psi, \forall\alpha.C) = \emptyset$$

ist, dann können wir G in G' und G'' aufteilen. Wenn nicht, dann nennen wir ein Element aus dem Schnitt α' und wenden die erste der beiden Regeln an. □

A.3.6 Erfüllbarkeitstest

Proposition 18 (geschlossene Constraints). *Ein geschlossener Typconstraint in Normalform $Q.P \wedge \psi \wedge G$ ist äquivalent zum Termconstraint $Q.P$*

Beweis:

In einem geschlossenen Typconstraint in Normalform ist $\mathrm{dom}(\psi) = \emptyset$ und daher $\psi = \mathrm{true}$. Es ist klar, daß $Q.P \wedge G \vdash^A Q.P$. Wenn Q in Typquantoren $\exists\bar{\beta}$ und Termquantoren Q_β aufgeteilt wird, so gilt die andere Richtung:

$$Q.P \vdash^A [\mathtt{int}/\bar{\beta}]Q_\beta(P \wedge G) \vdash^A \exists\bar{\beta}.Q_\beta.(P \wedge G) \vdash^A Q.P \wedge G$$

□

A.4 Operationelle Korrektheit

Lemma 4. *Es gilt für* $\bar{\gamma}, \bar{k} \notin \mathrm{fv}(C_5)$

$$\frac{C_0, \Gamma_x \vdash^W F : \nu \quad C_1, \Gamma \vdash^W E : \mu \quad C_5 \wedge C_4 \vdash^A \exists \nu.(C_0 \wedge \nu = \tau) \quad (x : \forall \bar{\gamma}.\forall \bar{k}.C_4 \Rightarrow \tau) \in \Gamma}{C_2, \Gamma_x \vdash^W [F/x]E : \mu}$$

und es gilt dann $\exists \nu.C_0 \wedge C_5 \wedge C_1 \vdash^A C_2$.

Beweis:
Wir beweisen dieses Lemma mit einer strukturellen Induktion über E. Der schwierige Fall ist die (var^W) Regel. Alle anderen Fälle folgen dem selben Schema. Wir haben jeweils eine letzte Regel

$$\frac{C_1, \Gamma \vdash^W \ldots}{C_1', \Gamma \vdash^W \ldots}.$$

Wir wenden die Induktionshypothese an und dann wieder die gleiche Regel

$$\frac{C_2, \Gamma \vdash^W \ldots}{C_2', \Gamma \vdash^W \ldots}.$$

Zu zeigen bleibt dann jeweils

$$\frac{\exists \nu.C_0 \wedge C_5 \wedge C_1 \vdash^A C_2}{\exists \nu.C_0 \wedge C_5 \wedge C_1' \vdash^A C_2'}.$$

Diese Beweise sind meist einfach.

- $$(var^W)\frac{(y : \forall \bar{\gamma}.\forall \bar{k}.C_4 \Rightarrow \tau) \in \Gamma}{\exists \bar{\gamma}.\exists \bar{k}.(C_4 \wedge \mu = \tau), \Gamma \vdash^W y : \mu} \quad \delta\, new$$

  Für $y \neq x$ gilt die Behauptung trivialerweise mit $C_2 = C_1$. Für $y = x$ haben wir durch Umbenennung

  $$[\mu/\nu]C_0, \Gamma_x \vdash^W [F/x]E : \mu.$$

  Wir zeigen

  $$\frac{C_5 \wedge C_4 \vdash^A \exists \nu.(C_0 \wedge \nu = \tau)}{\exists \nu.C_0 \wedge C_5 \wedge \exists \bar{\gamma}.\exists \bar{k}.(C_4 \wedge \mu = \tau) \vdash^A [\mu/\nu]C_0}$$

indem wir die untere Aussage unter einmaliger Verwendung der oberen herleiten.

$$\begin{array}{l}\exists\nu.C_0 \wedge C_5 \wedge \exists\bar{\gamma}.\exists\bar{k}.(C_4 \wedge \mu = \tau)\\ \quad \vdash^A \exists\nu.\exists\bar{\gamma}.\exists\bar{k}.C_5 \wedge C_4 \wedge \mu = \tau\\ \quad \vdash^A \exists\nu.\exists\bar{\gamma}.\exists\bar{k}.(\exists\nu.(C_0 \wedge \nu = \tau)) \wedge \mu = \tau\\ \quad \vdash^A \exists\nu.\exists\bar{\gamma}.\exists\bar{k}.(C_0 \wedge \nu = \tau \wedge \mu = \tau)\\ \quad \vdash^A \exists\nu.\exists\bar{\gamma}.\exists\bar{k}.(C_0 \wedge \nu = \mu)\\ \quad \vdash^A \exists\nu.([\mu/\nu]C_0 \wedge \nu = \mu)\\ \quad \vdash^A [\mu/\nu]C_0\end{array}$$

- $$(\rightarrow I^W)\frac{C, (\Gamma, y : \alpha) \vdash^W E : \beta}{\exists\alpha.\exists\beta.(\delta = \alpha \rightarrow \beta \wedge C), \Gamma \vdash^W \lambda y.E : \delta} \quad \alpha, \delta\, new$$

  Wir zeigen die am Anfang des Beweises charakterisierte Bedingung.

  $$\frac{\exists\nu.C_0 \wedge C_5 \wedge C_1 \vdash^A C_2}{\begin{array}{l}\exists\nu.C_0 \wedge C_5 \wedge \exists\alpha.\exists\beta.(\delta = \alpha \rightarrow \beta \wedge C_1)\\ \quad \vdash^A \exists\alpha.\exists\beta.(\delta = \alpha \rightarrow \beta \wedge C_2)\end{array}}$$

  Dies ist einfach zu sehen, da α und β nicht frei in $\exists\nu.C_0 \wedge C_5$ vorkommen.

- $$(\rightarrow E^W)\frac{C, \Gamma \vdash^W E : \alpha \quad C', \Gamma \vdash^W F : \beta}{\exists\alpha.\exists\beta.(C \wedge C' \wedge \alpha = \beta \rightarrow \delta), \Gamma \vdash^W EF : \delta} \quad \delta\, new$$

  Dieses Mal wenden wir die Induktionshypothese auf beide Prämissen an. Die Bedingung bleibt ähnlich und ist wieder leicht einzusehen.

  $$\frac{\exists\nu.C_0 \wedge C_5 \wedge C_1 \vdash^A C_2 \quad \exists\nu.C_0 \wedge C_5 \wedge C_1' \vdash^A C_2'}{\begin{array}{l}\exists\nu.C_0 \wedge C_5 \wedge \exists\alpha.\exists\beta.(C_1 \wedge C_1' \wedge \alpha = \beta \rightarrow \delta)\\ \quad \vdash^A \exists\alpha.\exists\beta.(C_2 \wedge C_2' \wedge \alpha = \beta \rightarrow \delta)\end{array}}$$

- $$(let^W)\frac{\begin{array}{c} C_1, \Gamma \vdash^W E : \alpha \\ C_1', (\Gamma, y : \forall\alpha.C_1 \Rightarrow \alpha) \vdash^W G : \beta \end{array}}{C_1' \wedge \exists\alpha.C_1, \Gamma \vdash^W (\textbf{let } y = E \textbf{ in } G) : \beta}$$

  Wir schließen $x = y$ aus, denn wir können diesen Fall durch Umbenennung vermeiden. Hier wäre er allerdings trivial.

  Mit der ersten Prämisse und der Induktionshypothese bekommen wir

  $$C_2, \Gamma \vdash^W [F/x]E : \alpha,$$

  wobei

  $$\exists\nu.C_0 \wedge C_5 \wedge C_1 \vdash C_2$$

  gilt. Somit gilt aber

  $$\exists\nu.C_0 \wedge C_5 \vdash^A (\forall\alpha.C_1 \Rightarrow \alpha) \sqsubseteq (\forall\alpha.C_2 \Rightarrow \alpha).$$

  Mit der Proposition von der Kontextabschwächung bekommen wir

  $$C_3', (\Gamma, y : \forall\alpha.C_2 \Rightarrow \alpha) \vdash^W G : \beta,$$

  wobei $\exists\nu.C_0 \wedge C_5 \wedge C_1' \vdash^A C_3'$ gilt. Nun wenden wir erneut die Induktionshypothese an und erhalten

  $$C_2', (\Gamma, y : \forall\alpha.C_2 \Rightarrow \alpha) \vdash^W [F/x]G : \beta$$

  mit $\exists\nu.C_0 \wedge C_5 \wedge C_3' \vdash^A C_2'$. Es folgt

  $$\frac{\exists\nu.C_0 \wedge C_5 \wedge C_1 \vdash^A C_2 \quad \exists\nu.C_0 \wedge C_5 \wedge C_1' \vdash^A C_2'}{\exists\nu.C_0 \wedge C_5 \wedge C_1' \wedge \exists\alpha.C_1 \vdash^A C_2' \wedge \exists\alpha.C_2}.$$

- $$(fix^W)\frac{\begin{array}{l} C, (\Gamma, x : \forall\bar{\alpha}.\forall\bar{n}.P \Rightarrow \tau) \vdash^W E : \gamma \\ \exists\bar{\alpha}.\exists\bar{n}.(\delta = \tau \wedge P \wedge \\ \quad \forall\bar{\alpha}.\forall\bar{n}.(P \rightarrow \exists\gamma.(C \wedge \tau = \gamma))), \\ \quad \Gamma \vdash^W (\textbf{fix } x :: P \Rightarrow \tau \textbf{ in } E) : \delta \end{array}}{\left\{ \begin{array}{l} \delta\, new \\ \bar{\alpha}, \bar{n} = \mathrm{fv}(P, \tau) \end{array} \right.}$$

Hier ist die Bedingung wieder einfach:

$$\frac{\exists\nu.C_0 \wedge C_5 \wedge C_1 \vdash^A C_2}{\begin{array}{l}\exists\nu.C_0 \wedge C_5 \wedge \exists\bar{\alpha}.\exists\bar{n}.(\delta = \tau \wedge P \wedge \\ \quad \forall\bar{\alpha}.\forall\bar{n}.(P \to \exists\gamma.(C_1 \wedge \tau = \gamma))) \\ \quad \vdash^A \exists\bar{\alpha}.\exists\bar{n}.(\delta = \tau \wedge P \wedge \\ \quad \forall\bar{\alpha}.\forall\bar{n}.(P \to \exists\gamma.(C_2 \wedge \tau = \gamma)))\end{array}}$$

- $$(\textbf{data } I^W)\frac{\texttt{data T}\bar{\alpha}\bar{n} = \sum_i D_i\bar{\tau}_i \Leftarrow Q_i, \bar{\beta}_i\bar{m}_i}{\exists\bar{\alpha}.\exists\bar{\beta}_i.\exists\bar{n}.\exists\bar{m}_i.(\delta = \bar{\tau}_i \to T\bar{\alpha}\bar{n} \wedge Q_i), \Gamma \vdash^W D_i : \delta}$$

  Die Behauptung ist trivial, da die Ersetzung $[F/x]D_i = D_i$ nichts verändert.

- $$(\textbf{data } E^W)\frac{\begin{array}{l}\texttt{data T}\bar{\alpha}\bar{n} = \sum_i D_i\bar{\tau}_i \Leftarrow Q_i, \bar{\beta}_i\bar{m}_i \\ C, \Gamma \vdash^W x : \eta \\ C_i, (\Gamma, \bar{y}_i : \bar{\rho}_i) \vdash^W E_i : \zeta_i\end{array}}{\begin{array}{l}\exists\eta.\exists\bar{\alpha}.\exists\bar{n}.(\eta = T\bar{\alpha}\bar{n} \wedge C \wedge \\ \quad \forall i.\forall\bar{\beta}_i.\forall\bar{m}_i.\exists\zeta_i.\exists\bar{\rho}_i.(Q_i \to \\ \quad (\bar{\rho}_i = \bar{\tau}_i \wedge C_i \wedge \zeta_i = \delta))), \\ \quad \Gamma \vdash^W \textbf{case } x \textbf{ of } \{D_i\bar{y}_i \Rightarrow E_i\} : \delta\end{array}} \quad \delta, \bar{\rho}_i \, new$$

  Auch im letzten Fall müssen wir die Bedingung überprüfen.

$$\frac{\exists\nu.C_0 \wedge C_5 \wedge C_1 \vdash^A C_2 \quad \exists\nu.C_0 \wedge C_5 \wedge C_{i1} \vdash^A C_{i2}}{\begin{array}{l}\exists\nu.C_0 \wedge C_5 \wedge \exists\eta.\exists\bar{\alpha}.\exists\bar{n}.(\eta = T\bar{\alpha}\bar{n} \wedge C_1 \wedge \\ \quad \forall i.\forall\bar{\beta}_i.\forall\bar{m}_i.\exists\zeta_i.\exists\bar{\rho}_i.(Q_i \to \\ \quad (\bar{\rho}_i = \bar{\tau}_i \wedge C_{i1} \wedge \zeta_i = \delta))) \\ \vdash^A \exists\eta.\exists\bar{\alpha}.\exists\bar{n}.(\eta = T\bar{\alpha}\bar{n} \wedge C_2 \wedge \\ \quad \forall i.\forall\bar{\beta}_i.\forall\bar{m}_i.\exists\zeta_i.\exists\bar{\rho}_i.(Q_i \to \\ \quad (\bar{\rho}_i = \bar{\tau}_i \wedge C_{i2} \wedge \zeta_i = \delta)))\end{array}}$$

□

Satz 4. *Es gilt*

$$\frac{C, \Gamma \vdash^W F : \alpha \quad F \to^* F'}{C', \Gamma \vdash^W F' : \alpha}$$

wobei $C \vdash C'$.

Beweis:

Wir zeigen zuerst

$$\frac{C, \Gamma \vdash^W F : \alpha \quad F \to F'}{C', \Gamma \vdash^W F' : \alpha}.$$

Wir betrachten dazu alle Reduktionen:

- $(\lambda x.E)F \to [F/x]E$

$$\frac{\dfrac{C_1, \Gamma, x : \delta \vdash^W E : \mu}{\exists\delta.\exists\mu.(C_1 \wedge \beta = \delta \to \mu), \Gamma \vdash^W (\lambda x.E) : \beta} \quad C_0, \Gamma \vdash^W F : \nu}{\exists\beta.\exists\nu.(\exists\delta.\exists\mu.(C_1 \wedge \beta = \delta \to \mu) \wedge C_0 \wedge \beta = \nu \to \alpha), \Gamma \vdash^W (\lambda x.E)F : \alpha}$$

  Wir wenden das Lemma auf die ersten beiden Zeilen an. Wir wählen $C_5 = \exists\nu.(C_0 \wedge \nu = \delta)$. Damit ist auch die dritte Voraussetzung direkt erfüllt. Mit dem Lemma gilt

$$\exists\nu.C_0 \wedge \exists\nu.(C_0 \wedge \nu = \delta) \wedge C_1 \vdash^A C_2.$$

  Um die Behauptung zu zeigen müssen wir jetzt noch

$$\begin{aligned}&[\mu/\alpha](\exists\beta.\exists\nu.(\exists\delta.\exists\mu.(C_1 \wedge \beta = \delta \to \mu) \wedge C_0 \wedge \beta = \nu \to \alpha))\\ &\quad \vdash^A [\mu/\alpha](\exists\beta.\exists\nu.(\exists\delta.\exists\mu.(C_1 \wedge \delta = \nu \wedge \mu = \alpha) \wedge C_0 \wedge \beta = \nu \to \alpha))\\ &\quad \vdash^A \exists\nu.(C_0 \wedge \nu = \delta) \wedge C_1\end{aligned}$$

  zeigen, was aber offensichtlich ist.

- $(\mathbf{fix}\ f :: \sigma\ \mathbf{in}\ E) \to [(\ \mathbf{fix}\ f :: \sigma\ \mathbf{in}\ E)/f]E$

$$(fix^W)\frac{\begin{array}{l}C_1, (\Gamma, x : \forall\bar{\gamma}.\forall\bar{k}.P \Rightarrow \tau) \vdash^W E : \mu\\ \exists\bar{\gamma}.\exists\bar{k}.(\nu = \tau \wedge P \wedge\\ \quad \forall\bar{\gamma}.\forall\bar{k}.(P \to \exists\mu.(C_1 \wedge \tau = \mu))),\\ \quad \Gamma \vdash^W (\mathbf{fix}\ x :: P \Rightarrow \tau\ \mathbf{in}\ E) : \nu\end{array}}{\left\{\begin{array}{l}\nu\ new\\ \bar{\gamma}, \bar{k} = \mathrm{fv}(P, \tau)\end{array}\right.}$$

Wir schreiben

$$\begin{aligned} C_0 = \exists\bar{\gamma}.\exists\bar{k}.(\nu = \tau \wedge P \wedge \\ \forall\bar{\gamma}.\forall\bar{k}.(P \rightarrow \exists\mu.(C_1 \wedge \tau = \mu))) \end{aligned}$$

für den unteren Constraint der Konklusion und wenden das Lemma mit

$$C_5 = \forall\bar{\gamma}.\forall\bar{k}.(P \rightarrow \exists\mu.(C_1 \wedge \tau = \mu)))$$

auf Antezedent und Konklusion an. Wir müssen noch die letzte Voraussetzung zeigen.

$$\begin{aligned} & C_5 \wedge P \\ & \quad \vdash^A C_5 \wedge \exists\nu.((\exists\bar{\gamma}.\exists\bar{k}.\nu = \tau \wedge P) \wedge \tau = \nu) \\ & \quad \equiv^A \exists\nu.((\exists\bar{\gamma}.\exists\bar{k}.\nu = \tau \wedge P \wedge C_5) \wedge \tau = \nu) \\ & \quad = \exists\nu.(C_0 \wedge \tau = \nu) \end{aligned}$$

Wir folgern mit dem Lemma

$$\exists\nu.C_0 \wedge C_5 \wedge C_1 \vdash C_2$$

und wollen

$$[\mu/\nu]C_0 \vdash^A C_2$$

zeigen, indem wir

$$[\mu/\nu]C_0 \vdash^A \exists\nu.C_0 \wedge C_5 \wedge C_1$$

zeigen. Es ist leicht einzusehen, daß $[\mu/\nu]C_0 \vdash^A \exists\nu.C_0 \wedge C_5$ gilt. $[\mu/\nu]C_0 \vdash^A C_1$ zeigen wir über

$$\begin{aligned} & [\mu/\nu]C_0 \\ & \quad \vdash^A [\mu/\nu](\exists\bar{\gamma}.\exists\bar{k}.(\nu = \tau \wedge P)) \wedge \\ & \qquad (\forall\bar{\gamma}.\forall\bar{k}.(P \rightarrow \exists\mu.(C_1 \wedge \tau = \mu))) \\ & \quad \vdash^A (\exists\bar{\gamma}.\exists\bar{k}.(\mu = \tau \wedge P)) \wedge \\ & \qquad (\forall\bar{\gamma}.\forall\bar{k}.(P \rightarrow \exists\mu.(C_1 \wedge \tau = \mu))) \\ & \quad \vdash^A \exists\bar{\gamma}.\exists\bar{k}.(\mu = \tau \wedge P \wedge \exists\mu.(C_1 \wedge \tau = \mu)) \\ & \quad \vdash^A C_1 \end{aligned}$$

herleiten.

- $(\textbf{let } x = E \textbf{ in } F) \to [E/x]F$

$$(let^W)\frac{\begin{array}{c} C_0, \Gamma \vdash^W E : \nu \\ C_1, (\Gamma, x : \forall\gamma.[\gamma/\nu]C_0 \Rightarrow \gamma) \vdash^W F : \mu \end{array}}{C_1 \wedge \exists\nu.C_0, \Gamma \vdash^W (\textbf{let } x = E \textbf{ in } F) : \mu}$$

Wir wenden wieder das Lemma auf die beiden oberen Aussagen an und wählen $C_5 = \text{true}$. Die letzte Voraussetzung gilt diesmal wegen

$$C_4 = [\gamma/\nu]C_0 \vdash^A \exists\nu.C_0 \wedge \nu = \gamma.$$

Aus dem Lemma folgt

$$\exists\nu.C_0 \wedge C_1 \vdash^A C_2,$$

aber das ist gerade, was wir zeigen mußten.

- $\textbf{case } (D\bar{E}) \textbf{ of } \{(D\bar{x}) \Rightarrow E'\} \to [\bar{E}/\bar{x}]E'$

Die Ableitung des Ausdrucks auf der linken Seite läßt sich auf die Verwendung folgender Regeln zurückführen.

Es beginnt mit der Einführung des Konstruktors D.

$$(\textbf{data } I^W)\frac{\texttt{data T}\bar{\alpha}\bar{n} = \ldots D\bar{\tau} \Leftarrow Q, \bar{\beta}\bar{m} \ldots}{\exists\bar{\alpha}.\exists\bar{\beta}.\exists\bar{n}.\exists\bar{m}.(\eta_0 = \bar{\tau} \to T\bar{\alpha}\bar{n} \wedge Q), \Gamma \vdash^W D : \eta_0}$$

Daran schließen sich eine Reihe von Applikationsregeln mit den Argumenten $\bar{F}$ an. Wir haben

$$(\to E^W)\frac{B_i, \Gamma \vdash^W (DF_1 \ldots F_i) : \eta_i \quad A_i, \Gamma \vdash^W F : \xi_i}{\begin{array}{c}\exists\eta_i.\exists\zeta_i.(B_i \wedge A_i \wedge \eta_i = \xi_i \to \eta_{i+1}), \Gamma \\ \vdash^W (DF_1 \ldots F_{i+1}) : \eta_{i+1}\end{array}},$$

wobei die B_i induktiv definiert sind.

$$B_0 = \exists\bar{\alpha}.\exists\bar{\beta}.\exists\bar{n}.\exists\bar{m}.(\eta_0 = \bar{\tau} \to T\bar{\alpha}\bar{n} \wedge Q)$$

$$B_{i+1} = \exists\eta_i.\exists\xi_i.(B_i \wedge A_i \wedge \eta_i = \xi_i \to \eta_{i+1})$$

Zuletzt folgt eine Anwendung der case-Regel:

$$(\textbf{data } E^W)\frac{\begin{array}{l}\texttt{data T}\bar{\alpha}\bar{n} = \ldots D\bar{\tau} \Leftarrow Q, \bar{\beta}\bar{m}\ldots \\ B_k, \Gamma \vdash^W (D\bar{F}) : \eta_k \\ H_k, (\Gamma, \bar{y} : \bar{\rho}) \vdash^W E : \zeta\end{array}}{\begin{array}{l}\exists\eta_k.\exists\bar{\alpha}.\exists\bar{n}.(\eta_k = T\bar{\alpha}\bar{n} \wedge B_k \wedge \\ \quad \forall\bar{\beta}.\forall\bar{m}.\exists\zeta.\exists\bar{\rho}.(Q \rightarrow \\ \quad (\bar{\rho} = \bar{\tau} \wedge H_k \wedge \zeta = \delta))), \\ \quad \Gamma \vdash^W \textbf{case } (D\bar{F}) \textbf{ of } \{D\bar{y} \Rightarrow E\} : \delta\end{array}}$$

Den Constraint des letzten Urteils bezeichnen wir wieder mit C_0:

$$\begin{array}{l}C_0 = \exists\eta_k.\exists\bar{\alpha}.\exists\bar{n}.(\eta_k = T\bar{\alpha}\bar{n} \wedge B_k \wedge \\ \quad \forall\bar{\beta}.\forall\bar{m}.\exists\zeta.\exists\bar{\rho}.(Q \rightarrow \\ \quad (\bar{\rho} = \bar{\tau} \wedge H_k \wedge \zeta = \delta)))\end{array}$$

Durch sukzessives Anwenden des Lemmas erhalten wir Urteile der Form

$$H_i, (\Gamma, y_1 : \rho_1 \ldots y_i : \rho_i) \vdash^W [F_{i+1}/x_{j+1} \ldots F_k/x_k]E : \zeta.$$

Wir zeigen

$$H_k \wedge \bigwedge_{j=i\ldots k} \exists\xi_j(A_j \wedge \xi_j = \rho_{j+1}) \vdash^A H_i$$

induktiv, indem wir das Lemma jeweils mit

$$C_5 = \exists\xi_{i-1}.(A_{i-1} \wedge \xi_{i-1} = \rho_i)$$

anwenden. Wir erhalten dann jeweils

$$\exists\xi_{i-1}.(A_{i-1} \wedge \xi_{i-1} = \rho_i) \wedge H_i \vdash H_{i-1}$$

und mit dem Induktionsanfang für $i = k$ gilt die Behauptung auch für alle kleineren i.

Wir müssen nun noch $C_0 \vdash^A [\delta/\zeta]H_0$ zeigen. Dazu behaupten wir zuerst allgemein

$$\begin{array}{l}C_0 \vdash^A \exists\eta_i.\exists\bar{\alpha}.\exists\bar{n}.\exists\xi_i \ldots \exists\xi_{k-1}.\eta_i = \xi_i \rightarrow \ldots \xi_{k-1} \rightarrow T\bar{\alpha}\bar{n} \\ \quad \wedge A_k \ldots A_i \wedge B_i \wedge \\ \quad \forall\bar{\beta}.\forall\bar{m}.\exists\zeta.\exists\bar{\rho}.(Q \rightarrow (\bar{\rho} = \bar{\tau} \wedge H_k \wedge \zeta = \delta))),\end{array}$$

was eine leichte Induktion von i nach $i-1$ durch Einsetzen von B_i

$$\begin{array}{l} C_0 \vdash^A \exists\eta_i.\exists\bar{\alpha}.\exists\bar{n}.\exists\xi_i \ldots \exists\xi_{k-1}.\eta_i = \xi_i \to \ldots \xi_{k-1} \to T\bar{\alpha}\bar{n} \\ \quad \wedge A_k \ldots A_i \\ \quad \wedge(\exists\eta_{i-1}.\exists\xi_{i-1}.(B_{i-1} \wedge A_{i-1} \wedge \eta_{i-1} = \xi_{i-1} \to \eta_i))\wedge \\ \quad \forall\bar{\beta}.\forall\bar{m}.\exists\zeta.\exists\bar{\rho}.(Q \to (\bar{\rho} = \bar{\tau} \wedge H_k \wedge \zeta = \delta))) \end{array}$$

und einige leichte Umformungen zeigen. Der Induktionsanfang $i = k$ ist aus der Definition von C_0 sofort ersichtlich.

Schließlich zeigen wir aus dem Fall für $i = 0$:

$$\begin{array}{l} C_0 \vdash^A \exists\eta_0\exists\bar{\alpha}.\exists\bar{n}.\exists\bar{\xi}.(\eta_0 = \bar{\xi} \to T\bar{\alpha}\bar{n} \wedge \bar{A} \wedge B_0 \\ \qquad \wedge\forall\bar{\beta}.\forall\bar{m}.\exists\zeta.\exists\bar{\rho}.(Q \to (\bar{\rho} = \bar{\tau} \wedge H_k \wedge \zeta = \delta))) \\ \quad \vdash^A \exists\eta_0\exists\bar{\alpha}.\exists\bar{n}.\exists\bar{\xi}.(\eta_0 = \bar{\xi} \to T\bar{\alpha}\bar{n} \wedge \bar{A} \\ \qquad \wedge\exists\bar{\alpha}.\exists\bar{\beta}.\exists\bar{n}.\exists\bar{m}.(\eta_0 = \bar{\tau} \to T\bar{\alpha}\bar{n} \wedge Q) \\ \qquad \wedge\forall\bar{\beta}.\forall\bar{m}.\exists\zeta.\exists\bar{\rho}.(Q \to (\bar{\rho} = \bar{\tau} \wedge H_k \wedge \zeta = \delta))) \\ \quad \vdash^A \exists\bar{\alpha}.\exists\bar{n}.\exists\bar{\xi}.\exists\bar{\beta}.\exists\bar{m}.((\bar{\xi} = \bar{\tau} \wedge \bar{A} \wedge Q) \\ \qquad \wedge\forall\bar{\beta}.\forall\bar{m}.\exists\zeta.\exists\bar{\rho}.(Q \to (\bar{\rho} = \bar{\tau} \wedge H_k \wedge \zeta = \delta))) \\ \quad \vdash^A \exists\bar{\alpha}.\exists\bar{n}.\exists\bar{\beta}.\exists\bar{m}.\exists\zeta.\exists\bar{\rho}.\exists\bar{\xi}. \\ \qquad (\bar{\xi} = \bar{\tau} \wedge \bar{A} \wedge \bar{\rho} = \bar{\tau} \wedge H_k \wedge \zeta = \delta) \\ \quad \vdash^A \exists\zeta.H_0 \wedge \zeta = \delta \\ \quad \vdash^A [\delta/\zeta]H_0 \end{array}$$

Damit ist

$$\frac{C, \Gamma \vdash^W F : \alpha \quad F \to F'}{C', \Gamma \vdash^W F' : \alpha}$$

gezeigt. Es ist offensichtlich daß diese Eigenschaft mit $\to$ auch für die reflexive, transitive Hülle gilt. Sie gilt aber aus folgendem Grund auch für die kompatible Hülle:

Nehmen wir an, wir haben

$$C, \Gamma \vdash^W E[F] : \alpha \quad F \to F'.$$

Dann haben wir irgendwo in der Ableitung von $C, \Gamma \vdash^W E[F] : \alpha$ das Urteil $C', \Gamma \vdash^W F$. Daraus können wir nun $C'', \Gamma \vdash^W F'$ folgern. Da aber $C' \vdash^A C''$ ist und die Abschwächung eines Constraints im Antezedent immer eine Abschwächung des Constraints in der Konklusion bedeutet gibt es nun ein

$$C''', \Gamma \vdash^W E[F'] : \alpha,$$

so daß $C \vdash^A C'''$.

□

Korollar 3 (Subjektreduktion). *Es gilt*

$$\frac{C, \Gamma \vdash F : \alpha \quad F \to^* F'}{C \wedge \mathrm{ext}(\Gamma), \Gamma \vdash F' : \alpha}$$

Beweis:
Sei $\sigma = \forall \bar{\alpha}.\forall \bar{n}.C'' \Rightarrow \tau$. Dann gilt wegen der Vollständigkeit

$$C', \Gamma \vdash^W F : \beta$$

wobei

$$C \wedge C'' \wedge \mathrm{ext}(\Gamma) \vdash^A \exists \beta.C' \wedge \beta = \tau.$$

Mit dem Satz von eben gilt

$$C''', \Gamma \vdash^W F : \beta$$

wobei $C' \vdash C'''$ und mit der Korrektheit gilt auch

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{C''', \Gamma \vdash F' : \beta}{C' \wedge \beta = \tau, \Gamma \vdash F' : \tau}}{\exists \beta.C' \wedge \beta = \tau, \Gamma \vdash F' : \tau}}{C \wedge C'' \wedge \mathrm{ext}(\Gamma), \Gamma \vdash F' : \tau}}{C \wedge \mathrm{ext}(\Gamma) \wedge \exists \bar{\alpha}.\exists \bar{n}.C'' \vdash F' : \forall \bar{\alpha}.\forall \bar{n}.C'' \Rightarrow \tau}}{C \wedge \mathrm{ext}(\Gamma) \wedge \mathrm{ext}(\sigma) \vdash F' : \sigma}}{C \wedge \mathrm{ext}(\Gamma) \vdash F' : \sigma}$$

Die letzte Zeile schließen wir aus der Konsistenz von $C, \Gamma \vdash F : \sigma$. □

A.5 Komplexität der Typüberprüfung

Definition 29 (Größe). *Wir definieren die Größe verschiedener Objekte:*

- *Termconstraints*

$$\begin{aligned}
|x| &= 1 \\
|f(t_1, \ldots, t_n)| &= 1 + \sum_i |t_i| \\
|\text{true}| &= 1 \\
|\text{indexfalse}| &= 1 \\
|t_1 \wedge t_2| &= |t_1| + |t_2| + 1 \\
|t_1 \vee t_2| &= |t_1| + |t_2| + 1 \\
|\neg t| &= |t| + 1 \\
|t_1 = t_2| &= |t_1| + |t_2| + 1 \\
|\exists n.t| &= |t| + 1 \\
|\forall n.t| &= |t| + 1
\end{aligned}$$

- *Typen*

$$\begin{aligned}
|\alpha| &= 1 \\
|\tau \to \tau'| &= |\tau| + |\tau'| + 1 \\
|T\bar{\tau}\bar{t}| &= 1 + \sum_i (|\tau_i| + |t|) \\
|\forall x.\sigma| &= |\sigma| + 1 \\
|P \Rightarrow \tau| &= |P| + |\tau| + 1
\end{aligned}$$

- *Kontexte*

$$\begin{aligned}
|(x_1 : \sigma_1, \ldots, x_n : \sigma_n)| &= 1 + \sum |\sigma_i| \\
|(x_1 : \sigma_1, \ldots, x_n : \sigma_n)|_{\text{m}} &= 1 + \max |\sigma_i|
\end{aligned}$$

- *Typdeklarationen*

$$
\begin{aligned}
|\mathbf{data}\ T\bar{\alpha}\bar{n} = \sum_i D_i\bar{\tau}_i \Leftarrow Q_i, \bar{\beta}_i\bar{m}_i| & \\
= \quad |\bar{\alpha}| + |\bar{n}| + \sum_i (|\bar{\tau}_i| + |Q_i| + |\bar{\beta}_i| + |\bar{m}_i|) & \\
|\Delta| \quad = \quad \sum_{d\in\Delta} |d| & \\
|\Delta|_{\mathrm{m}} \quad = \quad \max_{d\in\Delta} |d| &
\end{aligned}
$$

- *Typconstraints*

$$
\begin{aligned}
|\tau = \tau'| &= |\tau| + |\tau'| + 1 \\
|P \to C| &= |P| + |C| + 1 \\
|C \wedge C'| &= |C| + |C'| + 1 \\
|\forall\alpha.C| &= |C| + 1 \\
|\forall n.C| &= |C| + 1 \\
|\exists\alpha.C| &= |C| + 1 \\
|\exists n.C| &= |C| + 1
\end{aligned}
$$

- *Ausdrücke*

$$
\begin{aligned}
|x| &= 1 \\
|EF| &= |E| + |F| + 1 \\
|\lambda x.E| &= |E| + 1 \\
|\mathbf{let}\ x = E\ \mathbf{in}\ F| &= |E| + |F| + 1 \\
|\mathbf{fix}\ x :: P \Rightarrow \tau\ \mathbf{in}\ F| &= |F| + |P| + |\tau| + 1 \\
|D| &= 1 \\
|\mathbf{case}\ F\mathbf{of}\ \{D_i\bar{y}_i \Rightarrow E_i\}| &= |F| + 1 + \sum_i (|(D_i\bar{y}_i)| + |E_i|)
\end{aligned}
$$

Satz 5 (Komplexität). *Wenn ein funktionales Programm X von der Form*

$$\begin{array}{l}\textbf{let } f_1 = (\textbf{fix } f_1 :: P_1 \Rightarrow \tau_1 \textbf{ in } E_1) \textbf{ in}\\ \textbf{let } f_2 = (\textbf{fix } f_2 :: P_1 \Rightarrow \tau_2 \textbf{ in } E_2) \textbf{ in}\\ \vdots\\ \textbf{let } f_m = (\textbf{fix } f_m :: P_1 \Rightarrow \tau_m \textbf{ in } E_m) \textbf{ in}\\ F\end{array}$$

ist, Γ eine Kontext ohne freie Typvariablen ist, Δ die Menge der Typdeklarationen,

$$|P_i| < b, |\tau_i| < b, |\Delta|_{\mathrm{m}} < b, |E_i| < b, |F| < b, |\Gamma|_{\mathrm{m}} < 3 + 4b$$

und der Aufwand zur Entscheidung von $\vdash^A C$ für $|C| < b'$ durch $B'(b')$ beschränkt ist, dann ist der Aufwand zur Typüberprüfung mit $C, \Gamma \vdash X : \alpha$ durch $B(b)O((|X| + |\Gamma| + |\Delta|)\log(|X| + |\Gamma| + |\Delta|))$ beschränkt, also im wesentlichen linear in der Größe des Programms. Die Größe von C ist durch $B(b)$ beschränkt und damit in $O(1)$ entscheidbar.

Wir untersuchen nun den Aufwand für die Berechnung von C in $C, \Gamma \vdash^W E : \alpha$ und die Größe des entstehenden C. Der Aufwand zerfällt in zwei Teile, $T_S(C, \Gamma \vdash^W E : \alpha)$, für die Suche, Eintrage- und Löschoperationen von Variablen in Γ und Δ, und $T_R(C, \Gamma \vdash^W E : \alpha)$ für das rekursive Erstellen von C.

Die folgenden Lemmata machen Aussagen über T_S, T_R und die Größe des entstehenden C.

Lemma 5 (Aufwand: Suche). *Der Aufwand für die Suche $T_S(C, \Gamma \vdash^W E : \alpha)$ ist für beliebige Γ, Δ, E durch $O(|E|\log(|E| + |\Gamma| + |\Delta|))$ beschränkt.*

Beweis:
Wir betrachten eine Implementierung von Γ und Δ als balancierten Baum. Suche, Einfüge und Löschoperationen sind somit alle $O(|\Gamma|)$. Betrachten wir nun die Derivation von $C, \Gamma \vdash^W E : \alpha$. Die Zahl der Blätter, bzw. inneren Knoten dieses Derivationsbaums ist durch $|E|$ beschränkt. Die Größe des Kontextes Γ' in einem Blatt oder inneren Knoten ist durch $O(|E| + |\Gamma|)$ beschränkt. Also hat eine einzelne Suche, Einfüge oder Löschoperation den Aufwand $O(\log(|E|+|\Gamma|+|\Delta|))$. Da aber jedem Blatt maximal eine Suchoperation und jedem inneren Knoten maximal eine Einfüge und Löschoperation entsprechen, ergibt sich die gewünschte Abschätzung. □

Lemma 6 (Aufwand: allgemeine Ausdrücke). *Es gibt eine Konstante a, so daß für beliebige Γ, Δ, E*

$$T_R(C, \Gamma \vdash^W E : \alpha) < a^{|E|}(|\Gamma|_\mathrm{m} + |\Delta|_\mathrm{m} + 1)$$

und

$$|C| < a^{|E|}(|\Gamma|_\mathrm{m} + |\Delta|_\mathrm{m} + 1)$$

gilt.

Über den Aufwand zur Entscheidung von C wird hier nichts ausgesagt. Beweis:

Wir beweisen das Lemma mit struktureller Induktion. Wir führen die Beweise für $|C|$ nicht, da si genau denen für $T_R(C, \dots)$ entsprechen. O wird so verwendet, daß a nicht als Konstante zählt, also $a \notin O(1)$. Wir dürfen für a aber alle Ungleichungen verwenden, die für genügend große a gelten, also z.B. $a^2 > a$ oder $O(1) < a$.

$$T_R(C, \Gamma \vdash^W x : \alpha) < O(|\Gamma|_\mathrm{m}) < a(|\Gamma|_\mathrm{m}).$$

$$\begin{aligned}
&T_R(C, \Gamma \vdash^W EF : \alpha) \\
&\quad < \quad T_R(C', \Gamma \vdash^W E : \beta) + T_R(C'', \Gamma \vdash^W F : \delta) + O(1) \\
&\quad < \quad a^{|E|}(|\Gamma|_\mathrm{m} + |\Delta|_\mathrm{m} + 1) + a^{|F|}(|\Gamma|_\mathrm{m} + |\Delta|_\mathrm{m} + 1) + a \\
&\quad < \quad a^{|E|+|F|}(|\Gamma|_\mathrm{m} + |\Delta|_\mathrm{m} + 1).
\end{aligned}$$

$$\begin{aligned}
&T_R(C, \Gamma \vdash^W \textbf{let } x = E \textbf{ in } F : \alpha) \\
&\quad < \quad T_R(C', \Gamma \vdash^W E : \beta) + T_R(C'', (\Gamma, x : \forall\beta.C' \Rightarrow \beta) \vdash^W F : \delta) + O(1) \\
&\quad < \quad a^{|E|}(|\Gamma|_\mathrm{m} + |\Delta|_\mathrm{m} + 1) + a^{|F|}(|\Gamma, x : \forall\beta.C' \Rightarrow \beta|_\mathrm{m} + |\Delta|_\mathrm{m} + 1) + a \\
&\quad < \quad a^{|E|}(|\Gamma|_\mathrm{m} + |\Delta|_\mathrm{m} + 1) + a^{|F|}(|\Gamma|_\mathrm{m} + |C'| + |\Delta|_\mathrm{m} + 3) + a \\
&\quad < \quad a^{|E|}(|\Gamma|_\mathrm{m} + |\Delta|_\mathrm{m} + 1) + a^{|F|}(|\Gamma|_\mathrm{m} + |\Delta|_\mathrm{m} + 3) \\
&\qquad \quad + a^{|F|}a^{|E|}(|\Gamma|_\mathrm{m} + |\Delta|_\mathrm{m} + 1) + a \\
&\quad < \quad (a^{|E|} + 3a^{|F|} + a^{|E|+|F|} + a)(|\Gamma|_\mathrm{m} + |\Delta| + 1) \\
&\quad < \quad a^{|E|+|F|+1}(|\Gamma|_\mathrm{m} + |\Delta|_\mathrm{m} + 1).
\end{aligned}$$

$$
\begin{aligned}
&T_R(C, \Gamma \vdash^W \lambda x.E : \delta) \\
&\quad < \quad T_R(C', (\Gamma, x : \alpha) \vdash^W E : \beta) + O(1) \\
&\quad < \quad a^{|E|}(|\Gamma|_{\mathrm{m}} + |\Delta|_{\mathrm{m}} + 1) + a \\
&\quad < \quad a^{|E|+1}(|\Gamma|_{\mathrm{m}} + |\Delta|_{\mathrm{m}} + 1).
\end{aligned}
$$

$$
\begin{aligned}
&T_R(C, \Gamma \vdash^W (\mathbf{fix}\ x :: P \Rightarrow \tau\ \mathbf{in}\ E) : \delta) \\
&\quad < \quad T_R(C', (\Gamma, x : \forall\bar{\alpha}.\forall\bar{n}.P \Rightarrow \tau) \vdash^W E : \alpha) + O(|P \Rightarrow \tau|) \\
&\quad < \quad a^{|E|}(|\Gamma|_{\mathrm{m}} + 2|P \Rightarrow \tau| + |\Delta|_{\mathrm{m}} + 1) + a|P \Rightarrow \tau| \\
&\quad < \quad a^{|E|+|P\Rightarrow\tau|}(|\Gamma|_{\mathrm{m}} + |\Delta|_{\mathrm{m}} + 1).
\end{aligned}
$$

$$
\begin{aligned}
&T_R(C, \Gamma \vdash^W D_i : \delta) < O(|\Delta|_{\mathrm{m}}) < a|\Delta|_{\mathrm{m}} \\
&\quad < \quad a^{|D_i|}(|\Gamma|_{\mathrm{m}} + |\Delta|_{\mathrm{m}} + 1).
\end{aligned}
$$

$$
\begin{aligned}
&T_R(C, \Gamma \vdash^W \mathbf{case}\ F\ \mathbf{of}\ \{D_i\bar{y}_i \Rightarrow E_i\} : \delta) \\
&\quad < \quad T_R(C', \Gamma \vdash^W F : \eta) \\
&\qquad\quad +(\sum_i T_R(C_i, (\Gamma\bar{y}_i : \bar{\rho}_i) \vdash^W E_i : \zeta_i)) + O(|\Delta|_{\mathrm{m}}) \\
&\quad < \quad a^{|F|}(|\Gamma|_{\mathrm{m}} + |\Delta|_{\mathrm{m}} + 1) \\
&\qquad\quad +(\sum_i a^{|E_i|}(|\Gamma|_{\mathrm{m}} + |\Delta|_{\mathrm{m}} + 1)) + a|\Delta|_{\mathrm{m}} \\
&\quad < \quad a^{|\mathbf{case}\ F\ \mathbf{of}\ \{D_i\bar{y}_i\Rightarrow E_i\}|}(|\Gamma|_{\mathrm{m}} + |\Delta|_{\mathrm{m}} + 1).
\end{aligned}
$$

□

Lemma 7 (Aufwand: Ausdrücke der Form X). *Es gibt eine Konstante a', so daß für Γ und Δ, die den Anforderungen des Satzes genügen,*

$$T_R(C \wedge c, \Gamma \vdash^W X : \alpha) < a'(|X| + |\Gamma|_{\mathrm{m}} + |\Delta|_{\mathrm{m}})$$

und

$$|C| < a'$$

gelten. Dabei ist c true, false *oder* indexfalse.

Beweis:
Wir zeigen dies durch Induktion über die Struktur von X. Hier betrachten wir a und b als Konstanten, also O(1). a' darf vorerst nicht als solche betrachtet werden. Aber wie oben dürfen wir für a' alle Ungleichungen verwenden, die ab einer Mindestgröße von a' gelten, also z.B. $O(1) < a'$. Nach dem Lemma oben gilt:

$$\begin{aligned} T_R(C \wedge \text{true}, \Gamma \vdash^W F : \alpha) &< a^{|F|}(|\Gamma|_\text{m} + |\Delta|_\text{m} + 1) \\ &< \quad a^b(3 + 4b + b + 1) < O(1) < a' \end{aligned}$$

Für $|C|$ gilt analog $|C| < a'$.

Wir verfolgen den Typinferenzalgorithmus und die entstehenden Constraints unter der Annahme, daß der globale Typkontext Γ keine freien Variablen hat:

$$(*)\ \frac{\dfrac{\dfrac{C_\gamma, (\Gamma, f_1 : \sigma_1) \vdash^W E_1 : \gamma}{C_\beta, \Gamma \vdash^W (\textbf{fix}\ f_1 :: \sigma_1\ \textbf{in}\ E_1) : \beta}}{C'_\beta, \Gamma \vdash^W (\textbf{fix}\ f_1 :: \sigma_1\ \textbf{in}\ E_1) : \beta} \quad C_\delta \wedge c, (\Gamma, x : \forall\beta.C'_\beta \Rightarrow \beta) \vdash^W \ldots : \delta}{C_\delta \wedge c \wedge \exists\beta.C'_\beta, \Gamma \vdash^W \textbf{let}\ f_1 = \ldots\ \textbf{in}\ X' : \delta}$$

$$(*)\ \frac{C_\delta \wedge c \wedge \exists\beta.C'_\beta, \Gamma \vdash^W \textbf{let}\ f_1 = \ldots\ \textbf{in}\ X' : \delta}{C_\delta \wedge c', \Gamma \vdash^W \textbf{let}\ f_1 = \ldots\ \textbf{in}\ X' : \delta}$$

Die beiden mit $(*)$ markierten Regeln, bei denen geschlossene Teilconstraints vereinfacht werden. Diese *Optimierung* ist notwendig, um die Schranke zu zeigen.

Wir müssen zuerst zeigen, daß $|C'_\beta|$ klein genug ist, um die Induktionshypothese anzuwenden.

$$C_\beta = \exists\bar{\alpha}.\exists\bar{n}.(\beta = \tau \wedge P) \wedge \forall\bar{\alpha}.\forall\bar{n}.(P \rightarrow \exists\gamma.(C_\gamma \wedge \gamma = \tau))).$$

Der allquantifizierte Teilausdruck ist geschlossen, daher gilt

$$C_\beta \equiv C'_\beta = \exists\bar{\alpha}.\exists\bar{n}.(\beta = \tau \wedge P \wedge c'')).$$

Für die Größe gilt also

$$|C'_\beta| = |\bar{\alpha}| + |\bar{n}| + 1 + |\tau| + |P| < 1 + 4b.$$

Weiterhin ist β die einzig freie Variable in C_β. Also hat auch $\forall\beta.C'_\beta \Rightarrow \beta$ keine freie Variable, $|\forall\beta.C'_\beta \Rightarrow \beta| < 3+4b$ und wir können die Induktionshypothese anwenden:

Wir bekommen

$$|C_\delta| < a'$$

und

$$T_R(C_\delta \wedge c, \dots) < a'(|X| + |\Gamma|_\mathrm{m} + |\Delta|_\mathrm{m}).$$

Für die Größe von $|C_\delta|$ gilt nun offensichtlich die Behauptung.

Bei der Berechnung der Ausführungszeit, müssen wir nun auch den Aufwand für die Vereinfachungen mit einbeziehen.

Zuerst ist

$$|C_\beta| < O(|\sigma_1|) + |C_\gamma| < O(1) + a^{|E_1|}(|\Gamma|_\mathrm{m} + |\sigma_1| + |\Delta|_\mathrm{m} + 2) < O(1).$$

Damit bekommen wir

$$\begin{aligned} & T_R(C_\delta \wedge c', \dots) \\ < \;& T_R(C_\delta \wedge c \wedge \exists\beta.C'_\beta, \dots) + B'(|c \wedge \exists\beta.C'_\beta|) \\ < \;& T_R(C'_\beta, \dots) + T_R(C_\delta \wedge c, \dots) + O(1) \\ < \;& T_R(C_\beta, \dots) + B'(O(1)) + a'(|X| + |\Gamma|_\mathrm{m} + |\Delta|_\mathrm{m}) + O(1) \\ < \;& a'(|X| + O(1)). \end{aligned}$$

Wir beachten dabei, daß auch $c \wedge \exists\beta.C_\beta$ ein geschlossener Constraint ist, der mit dem Entscheidungsalgorithmus zu c'' zusammengefaßt werden kann.

□

Der Beweis des eigentlichen Satzes ist nun trivial. Beweis:
Aus den Lemmata sehen wir, daß

$$T_R(C, \Gamma \vdash^W X : \alpha) < a'(|X| + O(1))$$

und

$$T_S(C, \Gamma \vdash^W X : \alpha) < O(|X| \log(|X| + |\Gamma|_\mathrm{m} + |\Delta|_\mathrm{m}))$$

gilt. Daraus folgt sofort die Behauptung.

□

Index

Literaturverzeichnis

[Aug98] L. Augustsson. Cayenne – a language with dependent types. `http://www.cs.chalmers.se/~augustss`, 1998.

[Bar84] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics, Studies in Logic and the Foundations of Mathematics*. North-Holland, revised edition, 1984.

[Bar92] H.P. Barendregt. Lambda calculi with types. In S. Abramsky, D.M. Gabbai, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, pages 117–309. Oxford University Press, 1992.

[BH90] H.P. Barendregt and K. Hemerik. Types in lambda calculi and programming languages. Technical report, University Nijmegen and Eindhoven University of Technology, 1990.

[CAB+86] R.L. Constable, S.F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, J.T. Sasaki, and S.F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall, 1986.

[CCF+95] C. Cornes, J. Courant, J. Filiâtre, G. Huet, P. Manoury, C. Munôz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saïbi, and B. Werner. *The Coq Proof Assistant Reference Manual*. INRIA-Rocquencourt — CNRS - ENS Lyon, 5.10 edition, 1995.

[CH88] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[Chu32] A. Church. A set of postulates for the foundation of logic. *Annals of Math.*, 2(33–34):346–366, 1932.

[Chu40] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Col75] G.E. Collins. Quantifier elimination for real closed fields by cylindric algebraic decomposition. In *2nd GI Conf. on Automata Theory and Formal Languages*, LNCS 33, pages 134–183, Kaiserslautern, 1975. Springer.

[Coo72] D.C. Cooper. Theorem proving in arithmetic with multiplication. In B. Meltzer and D. Mitchie, editors, *Machine Intelligence 7*, pages 91–99. American Elsevier, New York, 1972.

[Cro93] R. L. Crole. *Categories for Types.* Cambridge University Press, 1993.

[Cur34] H.B. Curry. Functionality in combinatory logic. In *Proc. Nat. Acad. Science*, volume 20, pages 584–590, 1934.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17:471–522, 1985.

[DM82] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. 9th POPL*, pages 207–212, 1982.

[FM89] Y. Fuh and P. Mishra. Polymorphic subtype inference: Closing the theory-practice gap. In *TAPSOFT*, volume 73, pages 167–183, March 1989.

[FM90] Y. Fuh and P. Mishra. Type inference with subtypes. *Theoretical Computer Science*, 73:155–175, 1990.

[GE90] J. Grosch and H. Emmelmann. A tool box for compiler construction. In D. Hammer, editor, *Proc of the Third International Workshop on Compiler Compilers*, LNCS 477, pages 106–116, Schwerin, Oct 1990. Springer.

[Gir71] J. Girard. Une extension de l'interpretation de Gödel a l'analyse, et son application a l'elimination des coupures dans l'analyse et la theorie des types. In *2nd Scandinavian Logic Symposium*, pages 63–92, 1971.

[GJS96] J. Gosling, B. Joy, and G. Steele. The java language specification. Java Series, Sun Microsystems, 1996.

[GM94] C.A. Gunter and J.C. Mitchell, editors. *Theoretical Aspects of Object-Oriented Programming.* MIT Press, Cambridge, Massachusetts, 1994.

[GR89] A. Goldberg and D. Robson. *Smalltalk 80: The Language.* Addison Wesley, Reading, Massachusetts, 1989.

[Gun92] C. Gunter. *Semantics of Programming Languages.* MIT Press, Cambridge, Massachusetts, 1992.

[Hen93a] F. Henglein. Dynamic typing: Syntax and proof theory. TOPPS Report D-163, DIKU, University of Copenhagen, March 1993. Science of Computer Programming, Special Issue on European Symposium on Programming 1992 (to appear).

[Hen93b] F. Henglein. Type inference with polymorphic recursion. *Transactions on Programming Languages and Systems*, 15(2):253–289, Apr 1993.

[Hin69] R. Hindley. The principle type scheme of an object in combinatory logic. *Trans. Am. math. Soc.*, 146:29–60, Dec 1969.

[HL94] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proc. 21th ACM Symp. POPL*, pages 123–137, Portland, Oregon, 1994. ACM Press.

[How80] W.A. Howard. The formulae-as-types notion of construction. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays on combinatory logic, lambda calculus and formalism.* Academic Press, 1980.

[HPS96] J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *ACM Symposium on Principles of Programming Languages*, January 1996.

[Hug96] J. Hughes. Type specialisation for the λ-calculus; or, a new paradigm for partial evaluation based on type inference. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation*, LNCS 1110, pages 183–215, Heidelberg, Feb 1996. Dagstuhl, Germany, Springer.

[Jac91] B. Jacobs. *Categorical Type Theory.* Ph. D. dissertation, University of Nijmegen, 1991.

[Jac96] B. Jacobs. On cubism. *Journal of Functional Programming*, 6(3):379–391, May 1996.

[Jay98] C.B. Jay. Poly-dimensional regular arrays in FISH. `http://linus.socs.uts.edu.au/~cbj`, 1998.

[JGS93] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation.* Prentice Hall, New York, 1993.

[Jon94] M.P. Jones. *Qualified Types: Theory and Practice.* PhD thesis, University of Oxford, 1994.

[Jon95] M.P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(1):1–35, January 1995.

[Jon96] M.P. Jones. Using parameterized signatures to express modular structure. In *Proc. 23rd ACM Symposium on Principles of Programming Languages*, pages 68–78, St.Peteresburg Beach, Florida, Jan 1996. ACM Press.

[Jon97] M.P. Jones. First-class polymorphism with type inference. In *Proc. 24rd ACM Symposium on Principles of Programming Languages*, pages 483–496, Paris, Jan 1997. ACM Press.

[JS98] C.B. Jay and P.A. Steckler. The functional imperative: Shape! In *ESOP*, pages 139–153, 1998.

[Kae92] S. Kaes. Type inference in the presence of overloading, subtyping and recursive types. In *Proc. ACM Conf. Lisp and Functional Programming*, pages 193–204, 1992.

[KM89] P.C. Kanellakis and J.C. Mitchell. Polymorphic unification and ML typing. In *Proc. 16th ACM Symposium on Principles of Programming Languages*, pages 105–115, ??, Jan 1989. ACM Press.

[KR78] B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, New Jersey, 1978.

[KTU90] A.J. Kfoury, J. Tiuryn, and P. Urzyczyn. ML typability is DEXPTIME-complete. In A. Arnold, editor, *Proceedings of the CAAP*, volume 431 of *LNCS*, Copenhagen, Denmark, May 1990. Springer.

[KTU93] A.J. Kfoury, J. Tiuryn, and P. Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(1):290–311, April 1993.

[Läu92] K. Läufer. *Polymorphic Type Inference and Abstract Data Types*. PhD thesis, Department of Computer Science, New York University, July 1992.

[Läu96] K. Läufer. Type classes with existential types. *Journal of Functional Programming*, 6(3):485–517, May 1996.

[LP92] Z. Luo and R. Pollack. *LEGO Proof Development System: User's Manual*. Department of Computer Science, University of Edinburgh, 1992.

[Mat70] J. Matiyasevic. Diophantine representation of recursively enumerable predicates. In *Proc. of the Second Scandinavian Logic Symposium*, Amsterdam, 1970. North-Holland.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Mit90] J.C. Mitchell. Type systems for programming languages. In J. van Leeuwen, editor, *Formal Models and Semantics*, pages 365–458. Elsevier Science, Amsterdam, 1990.

[Mit91] J.C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(3):245–286, Jul 1991.

[Mit96] J.C. Mitchell. *Foundations for Programming Languages*. MIT Press, Massachusetts, 1996.

[ML84] P. Martin-Löf. *Intuitionistic Type Theory, Studies in Proof Theory*. Bibliopolis, Naples, 1984.

[MO84] A. Mycroft and R.A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23:295–307, 1984.

[MP88] J. Mitchell and G. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages*, 10(3):470–502, 1988.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.

[Myc84] A. Mycroft. Polymorphic type schemes and recursive definitions. In *Proc. 6th International Conference on Programming*, LNCS, 1984.

[NPS90] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin Löf's Type Theory. An introduction*. Oxford University Press, 1990.

[OL96] M. Odersky and K. Läufer. Putting type annotations to work. In *Proc. 23rd ACM Symposium on Principles of Programming Languages*, pages 54–67, St.Peteresburg Beach, Florida, Jan 1996. ACM Press.

[OSW98] M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 1998. (submitted).

[OW97] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proc. 24rd ACM Symposium on Principles of Programming Languages*, pages 146–159, Paris, Jan 1997. ACM Press.

[OWW95] M. Odersky, P. Wadler, and M. Wehr. A second look at overloading. In *Proc. ACM Conf. on Functional Programming and Computer Architecture*, pages 135–149, June 1995.

[Per90] N. Perry. *The Implementation of Practical Functional Programming Languages*. PhD thesis, Imperial College of Science, Technology, and Medicine, University of London, 1990.

[PHA+97] J. Peterson, K. Hammond, L. Augustsson, B. Boutel, W. Burton, J. Fasel, A. Gordon, J. Hughes, P. Hudak, T. Johnsson, M. Jones, E. Meijer, S. Peyton Jones, A. Reid, and P. Wadler. *Report on the Programming Language Haskell Version 1.4*, April 1997. `http://www.haskell.org`.

[PL89] F. Pfenning and P. Lee. Metacircularity in the polymorphic λ-calculus. *Theoretical Computer Science*, 89:137–159, 1989.

[PT98] L. Prechelt and W.F. Tichy. A controlled experiment to assess the benefits of procedure argument type checking. *IEEE Transactions on Software Engineering*, to appear, 1998.

[Pug92] W. Pugh. The omega test: A fast and practical integer programming algorithm for dependency analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

[Rém89] D. Rémy. Typechecking records and variants in a natural extension of ML. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*, Austin, Texas, Jan 1989.

[Rém92] D. Rémy. Typing record concatenation for free. In *Proc. 19th ACM Symp. POPL*, pages 166–176. ACM Press, 1992.

[Rém94] D. Rémy. Type inference for records in a natural extension of ML. In C.A. Gunter and J.C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming*, pages 67–95. MIT Press, Cambridge, Massachusetts, 1994.

[Rey74] J. Reynolds. Towards a theory of type structure. In *International Programming Symposium*, LNCS 19, pages 408–425. Springer, 1974.

[Rey94] J. Reynolds. Using category theory to design implicit conversions and generic operators. In C.A. Gunter and J.C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming*, pages 25–64. MIT Press, Cambridge, Massachusetts, 1994.

[Rob65] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–49, Jan 1965.

[RS92] M. Ryan and M. Sadler. Valuation systems and consequence relations. In S. Abramsky, Dov M. Gabbai, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, pages 2–78. Oxford University Press, Oxford, 1992.

[Sch24] M. Schönfinkel. Über die Bausteine mathematischer Logik. *Math. Ann.*, 92:305–316, 1924.

[Sco76] D.S. Scott. Data types as lattices. *SIAM Journal on Computing*, 5:522–587, 1976.

[Sco82] D. S. Scott. Domains for denotational semantics. In *ICALP*, pages 577–613, 1982.

[SS78] G.L. Steele and G.J. Sussmann. The revised report on scheme. AI Memo 452, MIT, Jan 1978.

[Str67] C. Strachey. Fundamental concepts in programming languages. In *Lecture Notes for International Summer School in Computer Programming*, Copenhagen, Aug 1967.

[Sul96] M. Sulzmann. Typinferenz mit Constraints (german). Master's thesis, University of Karlsruhe, Mai 1996.

[Tur82] D.A. Turner. Recursion equations as a programming language. In Darlington et al., editor, *Functional Programming and Its Applications*. Cambridge University Press, 1982.

[Wad87] P. Wadler. List comprehensions. In S. Peyton Jones, editor, *The Implementation of Functional Programming Languages*, pages 127–138. Prentice-Hall International, Englewood Cliffs, New Jersey, 1987.

[Wan87a] M. Wand. Complete type inference for simple objects. In *Proc. Second IEEE Symposium on Logic in Computer Science*, pages 37–44, 1987.

[Wan87b] M. Wand. A simple algorithm and proof for type inference. *Fundamenta Informaticae*, 10:115–122, 1987.

[WB89] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *16th ACM Symposium on Principles of Programming Languages*, pages 60–76, 1989.

[Weh97] M. Wehr. *Überladung in Typsystemen.* PhD thesis, Universität Karlsruhe, Oct 1997.

[Wel94] J.B. Wells. Typability and type checking in the second order λ-calculus are equivalent and undecidable. In *Proc. 9th IEEE Symposium on Logic in Computer Science*, pages 176–185, July 1994.

[Wir90] M. Wirsing. Algebraic specification. In Jan van Leeuwen, editor, *Formal Methods and Semantics, Handbook of Theoretical Computer Science*, pages 675–788. Elsevier, 1990.

[WWRT91] J. Wileden, A. Wolf, W. Rosenblatt, and P. Tarr. Interoperability. *Communications of the ACM*, 34(5):73–87, May 1991.

[Xi98] H. Xi. *Dependent Types in Practical Programming.* PhD thesis, Department of Mathematical Sciences, Carnegie Mellon University, 1998.

[XP98] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *PLDI*, pages 249–257, 1998.

[Zen97] C. Zenger. Indexed types. *Theoretical Computer Science*, 187:147–165, 1997.

Lebenslauf

- Geboren am 28.Juli 1966 als Sohn von Christof und Elisabeth Zenger in München.
- 1972–1985 Besuch des Gymnasiums in Haar.
- 1985–1986, 1987–1989 und 1990–1993 Studium der Informatik an der Universität Karlsruhe mit Abschluß Diplom.
- 1986–1987 Wehrdienst in München.
- 1989–1990 Studium der Informatik am Weizmann Institute of Science in Rehovot, Israel.
- seit 1993 Wissenschaftlicher Mitarbeiter bei Prof. Calmet am Institut für Algorithmen und Kognitive Systeme der Fakultät Informatik der Universität Karlsruhe.