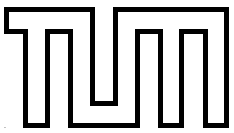


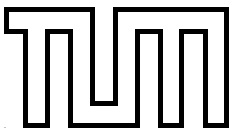
Technische Universität München

Fakultät für Mathematik

Bachelorarbeit in Mathematik

Beweis der Äquivalenz von Auswertung durch Substitution und Auswertung mit Umgebungen von λ-Termen in Isabelle

Armin Heller



Technische Universität München

Fakultät für Mathematik

Bachelorarbeit in Mathematik

Beweis der Äquivalenz von Auswertung durch Substitution und Auswertung mit Umgebungen von λ-Termen in Isabelle

A proof of the equivalence of reduction of λ-terms using substitution and reduction using environments, in Isabelle

Bearbeiter:	Armin Heller
Aufgabensteller:	Prof. Dr. Tim Hoffmann
Betreuer:	Dr. Christian Urban
Abgabedatum:	15. September 2010

Ich versichere, dass ich diese Bachelorarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15. September 2010

.....
(Unterschrift des Kandidaten)

Zusammenfassung

In dieser Arbeit geht es um die Theorie zweier Implementierungen des λ-Kalküls. Eine Implementierung benutzt Variablennamen, die andere benutzt stattdessen deBruijn-Indizes. Für beide Implementierungen werden "call-by-value"-Auswertungsrelationen definiert. Die Auswertungsrelation in der Implementierung mit deBruijn-Indizes benutzt Umgebungen, während die andere Auswertungsrelation Substitution verwendet. Desweiteren wird eine Übersetzungsrelation definiert, welche die λ-Terme beider Implementierungen ineinander überführt. Diese Übersetzung findet sich sinngemäß in vielen Compilern und Theorembeweisern wieder, ist jedoch selten Gegenstand von Beweisen. In dieser Arbeit wird die Korrektheit eben dieser Übersetzung gezeigt. Die Implementierung sowie der Korrektheitsbeweis sind im Theorembeweiser Isabelle entwickelt worden, was Fehler in den Lemmata und in deren Beweisen unmöglich macht.

Abstract

This thesis is about the theory of two implementations of the λ-calculus. The first implementation uses variable names, the second implementation uses deBruijn-indices instead. For both implementations a definition for "call-by-value" reduction is be given. The reduction of the deBruijn-implementation makes use of environments while the other reduction makes use of substitution. Furthermore a compilation relation is defined, which converts λ-terms between the two implementations. The concept of this translation can be found in many compilers and theorem provers, although proofs about it are rare. A proof of correctness of this translation is given in this thesis. The implementation and the proof have been developed with the theorem prover Isabelle. That's why errors in the lemmas and proofs of this thesis are impossible.

Inhaltsverzeichnis

1 Einleitung

Der λ-Kalkül ist ein formales System, dessen Grundidee die Beschreibung namenloser Funktionen ist, wie zum Beispiel $x \mapsto f\ (x)$. In diesem Beispiel finden sich schon die drei Grundelemente der Term-Sprache des λ-Kalküls: Variablen, wie x und f, die Abstraktion, hier durch $x \mapsto$ gekennzeichnet, und die Funktionsanwendung, die Applikation $f\ (x)$. Die Abstraktion wird historisch bedingt im λ-Kalkül anders geschrieben als in der Mathematik, nämlich mit dem griechischen Buchstaben λ und einem Punkt. Die Applikation benötigt keine Klammern. Das Beispiel sieht in dieser Notation wie folgt aus: $\lambda\ x.\ f\ x$.

Der λ-Kalkül kann benutzt werden, um damit höherstufige Logiken zu entwickeln. Höherstufige Logiken sind Grundlagen für Theorembeweiser und lassen sich als Fundierung für einen Großteil der Mathematik benutzen. Beispiele für höherstufige Logiken finden sich etwa in [7]. Konkreter dient der λ-Kalkül zur Fundierung des Theorembeweisers Isabelle, siehe [4]. Wie der λ-Kalkül benutzt werden kann, um darauf Programmiersprachen aufzubauen, wird in [6] gezeigt.

Als Berechenbarkeits-Modell benötigt der λ-Kalkül Regeln, nach denen die λ-Terme ausgewertet werden. Diese Regeln entsprechen meistens dem Einsetzen eines Arguments für den Parameter einer Funktion. Erlaubt man dieses Einsetzen überall, so erhält man die (nichtdeterministische) β-Reduktion. Legt man die Reihenfolge der Einsetzungen so weit fest, dass jeweils höchstens eine Einsetzung in einem Term erlaubt wird, so erhält man eine (deterministische) Auswertungs-Strategie. Die wichtigsten Auswertungs-Strategien sind "call-by-value" und "call-by-need". Diese finden sich in Programmiersprachen wieder, die auf dem λ-Kalkül aufbauen. ("call-by-need": Haskell, Clean; "call-by-value": SML, OCaml) In dieser Arbeit geht es nur um die "call-by-value"-Auswertung, jedoch in zwei verschiedenen Varianten, einmal definiert mit Hilfe von Substitution, einmal mit Hilfe von Umgebungen. Der formale Beweis, dass beide Varianten das gleiche leisten, ist das Ziel der Arbeit.

Die in dieser Arbeit angegebenen Lemmata sind alle im Theorembeweiser Isabelle formal bewiesen. Der Lesbarkeit halber werden die technischen Details, also der "Beweis-Code", oft weggelassen. Eine Einführung in Isabelle findet man in [3].

Die Formalisierung in Isabelle benutzt Nominal. Nominal ist eine Bibliothek an Lemmata, Definitionen und Taktiken (Algorithmen zur Beweisfindung), welche die Entwicklung von Theorien über Sprachen ermöglicht, in denen es das Konzept der gebundenen Variablen gibt. Gebundene Variablen entstehen beispielsweise durch Quantoren wie $\forall x.t$ oder $\exists x.t$, durch Auswahl-Operatoren wie $THE\ x.\ t$, oder eben durch die Abstraktion $\lambda x.\ t$ des λ-Kalküls. Der Name x ist in all diesen Beispielen eine gebundene Variable, und kann auch in t wieder auftreten. Umbenennung von gebundenen Variablen soll die Bedeutung eines Terms nicht verändern, da eine gebundene Variable nur als Platzhalter angesehen wird. Das x in $\forall x.\ t$ soll etwa ein Platzhalter sein für ein beliebiges Objekt, von welchem die Formel dann die Aussage t behauptet. Da Umbenennung von gebundenen Variablen die Bedeutung erhalten soll, kann

man in Beweisen davon ausgehen kann, dass gebundene Variablen immer so umbenannt wurden, dass sie andere Namen haben als ungebundene Variablen und dass verschiedene Quantoren oder Abstraktionen auch verschiedene Namen binden. Diese Vereinbarung ist Barendregts Variablenkonvention und hat in formalen Beweisen für Komplikationen gesorgt, die zum Glück durch Nominal gelöst wurden. Eine Erklärung der Lösung findet man in [2]. Was Bindungsstrukturen in Nominal sind,und welche Techniken für sie von Nominal zur Verfügung gestellt werden, ist in [1] beschrieben.

2 Verwandte Arbeiten

Die Idee für diese Arbeit stammt aus der Abhandlung "Compiler Verification in LF" [5]. Jene Veröffentlichung hat das gleiche Ziel wie diese. In [5] wird eine Formalisierung in LF (logical framework) entwickelt, hier soll eine Formalisierung in Isabelle entwickelt werden. Daraus ergeben sich erhebliche Unterschiede.

LF ist eine Typ-Theorie, die ebenfalls auf dem λ-Kalkül aufbaut, und wird zum Beispiel in [8] beschrieben. In LF lassen sich unter anderem Programmiersprachen und logische Systeme definieren und analysieren, sowie beispielsweise der λ-Kalkül selbst. Während Isabelle ein generischer Theorembeweiser mit einer festen Logik ist, wird LF als Meta-Logik dazu benutzt, um für spezielle Anwendungen spezielle Objekt-Logiken zu entwickeln.

Der wichtigste Unterschied zu [5] ist, dass Substitution in [5] nicht explizit definiert wird. Stattdessen wird das Einsetzen der λ-Terme der Objekt-Logik durch die Applikation in der Meta-Logik realisiert. Einerseits werden dadurch die Korrektheits-Beweise einfach, da man über die Substitution nichts beweisen muss, andererseits bedeutet das natürlich, dass das Resultat schwächer ist, weil es nur etwas über Funktionen in LF aussagt, nicht aber über Bindungsstrukturen.

Während in [5] nur der Beweis für das Lemma der Korrektheit (in [5] "soundness" genannt) ausgeführt wird, sind hier die Korrektheit und die Vollständigkeit (in [5]: "completeness") beide in Kapitel 4 bewiesen. Auch die Definition der Übersetzungsrelation unterscheidet sich erheblich, da Definitionen in LF sich nicht ohne weiteres in Isabelle übertragen lassen. Desweiteren werden hier noch andere Eigenschaften der Übersetzung bewiesen, die von Korrektheit und Vollständigkeit nicht abgedeckt werden.

3 Definitionen

3.1 Begriffs-Arten

In diesem Abschnitt werden die Begriffe definiert, die benötigt werden, um über λ-Terme Beweise führen zu können. Es handelt sich um drei Arten von Begriffen:

- Datentypen
- Funktionen
- Relationen

Induktive Datentypen sind eine Verallgemeinerung von bekannten mathematischen Strukturen wie den natürlichen Zahlen oder Tupeln. Isabelle basiert auf einer höherstufigen Logik (HOL). In dieser Logik gibt es Terme, z.B. $2 + x$, und es gibt Typen, z.B. $nat \Rightarrow nat$. Terme und Typen können miteinander in Relation stehen, z.B. $2 :: nat$. Intuitiv entspricht ein Typ einer Menge, und Terme können Elemente dieser Mengen sein. Der Operator :: entspricht dann der Element-Relation $\in$. Die Beziehung $2 :: nat$ bedeutet also, dass 2 ein Element der natürlichen Zahlen ist. Funktionen haben immer einen Typ $'a \Rightarrow 'b$, wobei $'a$ und $'b$ wieder Typen sind.

Eine bestimmte Art von Typen sind die induktiven Datentypen, wie es sie z.B. in der Programmiersprache OCaml gibt. Am besten versteht man induktive Datentypen, indem man sich ein paar Definitionen bekannter Datentypen ansieht:

datatype *Bool* =
  true
| *false*

datatype *Point2D* =
  Point int int

datatype *Nat* =
  Zero
| *Succ Nat*

Allgemein hat eine Definition eines induktiven Datentyps folgende Struktur:

datatype *type_name* =
  $C_1\ \alpha_{1,1} \ldots \alpha_{1,n_1}$
| $C_2\ \alpha_{2,1} \ldots \alpha_{1,n_2}$
$\vdots$
| $C_m\ \alpha_{m,1} \ldots \alpha_{m,n_m}$

Dabei sind die $\alpha_{i,j}$ Typen, die auch *type_name* enthalten können, und die C_k Konstruktoren. Ein Wert dieses Datentyps hat die Form $C_k\ x_{k,1}\ \ldots\ x_{k,n_k}$, wobei $x_{k,j} :: \alpha_{k,j}$ gelten muss.

Zwei Werte des selben induktiven Datentyps sind genau dann gleich, wenn sie in ihren Konstruktoren übereinstimmen und alle Argumente der Konstruktoren gleich sind.

Mit der Definition eines induktiven Datentyps erhält man ein Induktionsschema, welches es vereinfacht, Beweise entlang der Struktur der Definition zu führen. Bei *Bool* reduziert sich das Schema auf eine Fallunterscheidung. Bei *Point2D* ist das Schema einfach das einsetzen der Definition und bei *Nat* ergibt sich das bekannte Schema der vollständigen Induktion entlang der natürlichen Zahlen. Wie man sieht sind induktive Datentypen in Isabelle sehr flexibel. Außerhalb von Isabelle finden sie in funktionalen Programmiersprachen wie OCaml, SML oder Haskell Anwendung.

Funktionsdefinitionen sehen in Isabelle beispielsweise wie folgt aus:

fun *fib* :: *nat* ⇒ *nat* **where**
  fib 0 = *0*
| *fib* (*Suc 0*) = *1*
| *fib* (*Suc* (*Suc n*)) = *fib* (*Suc n*) + *fib n*

Damit werden die Fibonacci-Zahlen definiert. *Suc* ist die Nachfolger-Operation auf *nat*, den in Isabelle vordefinierten natürlichen Zahlen.

Allgemein sieht eine Funktions-Definition in Isabelle wie folgt aus:

fun $fn\_name :: fn\_type$ **where**
  $fn\_name\ pattern_1 = case_1$
| $fn\_name\ pattern_2 = case_2$
⋮
| $fn\_name\ pattern_n = case_n$

Dabei darf in den $case_i$ wieder fn_name vorkommen. Hat eine Funktion mehrere Argumente, so werden diese in $pattern_i$ einfach durch Leerzeichen getrennt, nicht durch Kommata, z.B. *f x y z*.

Ganz ähnlich sind die induktiven Relations-Definitionen aufgebaut. Beispiel:

inductive *even* :: *nat* ⇒ *bool* **where**
  even 0
| *even n* ⟹ *even* (*Suc* (*Suc n*))

Relationen müssen immer einen Funktionstyp haben, dessen Ergebnis *bool* ist. Allgemein sehen solche Relationsdefinitionen so aus:

inductive $rel\_name :: rel\_type$ **where**
  $hyps_1 \Longrightarrow rel\_name\ args_1$
| $hyps_2 \Longrightarrow rel\_name\ args_2$
⋮
| $hyps_n \Longrightarrow rel\_name\ args_n$

Wieder darf *rel_name* in den $hyps_i$ vorkommen. Liegen mehrere Voraussetzungen H_k in $hyps_i$ vor, so wird $hyps_i$ so geschrieben: $[\![H_1\ ;\ \ldots\ ;\ H_n]\!]$.

Bei allen drei Definitions-Arten entstehen gewisse Anforderungen an die Wohlgeformtheit der Definition, zum Beispiel muss die rekursive Definition einer Funktion immer terminieren. Diese Wohlgeformtheit wird meistens von Isabelle im Hintergrund automatisch bewiesen.

Die hier präsentierten Schemata für die Definitionen sind noch nicht die allgemeinst möglichen Schemata. Für die vom Author getätigten Definitionen reichen die Schemata allerdings aus.

Zwei wichtige vordefinierte Datentypen sollen hier noch vorgestellt werden: Listen und Paare.

Listen könnte man informell so definieren:

datatype $'a\ list =$
$[]$
$|\ 'a\ \#\ 'a\ list$

Dabei sind [] und # die Konstruktoren, das # ist der Lesbarkeit halber als infix-Operator geschrieben. Die Listendefinition ist schematisch polymorph mit dem Typ-Parameter $'a$, was bedeutet, dass alle Elemente einer Liste den gleichen Typ haben müssen, verschiedene Listen jedoch Elemente von verschiedenen Typen haben können. Listen werden in dieser Arbeit meistens mit einem s am Namensende bezeichnet, wie xs, ys oder $hyps$.

Die hier wichtigste Funktion, die mit Listen arbeitet, ist *map*:

fun $map :: ('a \Rightarrow 'b) \Rightarrow 'a\ list \Rightarrow 'b\ list$ **where**
$map\ f\ [] = []$
$|\ map\ f\ (x\ \#\ xs) = f\ x\ \#\ map\ f\ xs$

Die Liste $map\ f\ xs$ ist das Bild der Liste xs unter der Funktion f.

Tupel könnten informell so definiert werden:

datatype $'a\ \times\ 'b = ('a,\ 'b)$

Der Typ eines Tupels ist $'a\ \times\ 'b$, der Typ-Konstruktor $\times$ wird infix geschrieben. Als Menge gesehen entspricht der Typ $'a\ \times\ 'b$ der Produktmenge aus $'a$ und $'b$. Die beiden Komponenten eines Tupels können verschiedene Typen haben. Tupel werden wie üblich mit runden Klammern außen und einem infix-Komma geschrieben.

Zwei Funktionen zum Zerlegen eines Tupels sind fst und *snd*. Sie extrahieren die erste bzw. zweite Komponente eines Tupels:

fun $fst :: 'a \times 'b \Rightarrow 'a$ **where**
$fst\ (x,\ y) = x$

fun $snd :: 'a \times 'b \Rightarrow 'b$ **where**
$snd\ (x,\ y) = y$

3.2 λ-Terme

Die intuitive Definition von λ-Termen ist die folgende:

datatype *lam* =
  V name Variable
| *lam* $ *lam* Applikation
| λ *name*. *lam* Abstraktion

Bei dieser Definition sind V, \$ und λ die Konstruktoren. Der Lesbarkeit zuliebe wird \$ als infix-Operator geschrieben und ein Punkt nach dem Namen in der Abstraktion gesetzt. λ-Terme werden in dieser Arbeit meistens mit s, t oder r bezeichnet. Es soll abzählbar unendlich viele Namen vom Typ *name* geben. Namen werden mit x, y, z oder v bezeichnet. Die Vorkommen eines Namens x in $\lambda x.t$ zwischen dem λ und dem Punkt, und im folgenden t, werden als gebunden bezeichnet. Der Name x kommt in $\lambda x.Vy\$Vx$ zweimal gebunden vor, und y kommt nicht gebunden vor.

Um zu verstehen, warum diese Definition im folgenden nicht verwendet wird, ist es nötig, den Begriff der α-Äquivalenz zumindest informell zu erklären.
$s \approx_\alpha t$ soll genau dann gelten, wenn s und t sich durch Umbenennung von gebundenen Variablen ineinander überführen lassen. Beispielsweise gilt: $\lambda x.x \approx_\alpha \lambda y.y$. Die α-Äquivalenz ist eine Äquivalenzrelation.

Das explizite Behandeln von α-Äquivalenz ist aufwendig. Daher betrachtet man lieber direkt einen Datentyp, der einer α-Äquivalenzklasse solcher λ-Terme entspricht. Die oben definierten λ-Terme heißen ab jetzt λ-Bäume.

Hier nun die eigentliche Definition der λ-Terme mit Nominal:

nominal_datatype lam =
  V name
| *lam* $ *lam*
| Λ «*name*». *lam*

Die Zeichen «und »zeigen dabei an, dass Namen hinter Λ ab jetzt nur zur Repräsentation von α-Äquivalenzklassen dienen.

Eine zweite Möglichkeit, die explizite Behandlung der α-Äquivalenz zu umgehen, besteht darin, deBruijn-Indizes zu verwenden. Ein deBruijn-Index i ist eine natürliche Zahl, die in einem Term-Kontext angibt, dass man in der Term-Struktur i Abstraktionen "nach außen gehen" muss, um zu der Abstraktion zu kommen, die den Index i bindet.
Die daraus resultierende Term-Definition ist die folgende:

datatype *db_lam* =
  Var nat
| *Abs db_lam*
| *App db_lam db_lam*

Man beachte, dass bei der Verwendung von deBruijn-Indizes die Abstraktion keinen Namen

oder Index benötigt. Das Präfix *db_* steht für deBruijn.

Die Datentypen *lam* und *db_lam* sind die wichtigsten Datentypen in dieser Arbeit. Es geht im Folgenden nur um deren Zusammenspiel.

3.3 Permutationen und freie Variablen

In diesem Abschnitt werden die λ-Terme vom Typ *lam* verwendet.

Eine Permutation hat hier den Typ $(name \times name)\ list$. Sie repräsentiert eine bijektive Funktion, die nur auf endlich vielen Werten nicht der Identität entspricht, durch eine Liste.

Der Operator $\cdot$ wandelt die Listen-Repräsentation in die eigentliche Funktion vom Typ $name \Rightarrow name$ um. Permutationen werden hier meist mit *pi* bezeichnet.

```
fun _·_ :: (name × name) list ⇒ name ⇒ name where
  [] · z = z
| ((x, y) # pi) · z =
    (if x = pi · z
     then y
     else if y = pi · z
     then x
     else pi · z)
```

Permutationen können nicht nur auf Namen angewandt werden, sondern auch auf Listen.

```
fun _·_ :: (name × name) list ⇒ name list ⇒ name list where
  pi · [] = []
| pi · (x # xs) = (pi · x) # (pi · xs)
```

Zu guter Letzt werden Permutationen noch für λ-Terme benötigt:

```
nominal_primrec _·_ :: (name × name) list ⇒ lam ⇒ lam where
  pi · (V x) = V (pi · x)
| pi · (s $ t) = (pi · s) $ (pi · t)
| pi · (Λ x. t) = Λ (pi · x). (pi · t)
```

Ebenfalls im folgenden oft verwendet wird der Begriff der freien Variablen:

```
nominal_primrec free_vars :: lam ⇒ name set where
  free_vars (V x) = {x}
| free_vars (s $ t) = free_vars s ∪ free_vars t
| free_vars (Λ x. t) = (free_vars t) − {x}
```

Die Tatsache, dass ein Name nicht frei in einem Term vorkommt, drückt der folgende Begriff aus:

```
definition _♯_ :: name ⇒ lam ⇒ bool where
(x ♯ t) = x ∉ free_vars t
```

Analog zu Permutationen ist $\sharp$ auch für Namen, Listen von Namen und Tupel definiert.

Die Definitionen aus diesem Abschnitt müssen in Isabelle nicht getätigt werden, da das Nominal-Paket die definierten Begriffe und viele Lemmata bereits zur Verfügung stellt. Alle Definitionen dieses Abschnittes dienen nur der Anschauung und haben im Beweis-Code keine Bedeutung.

3.4 Substitution

Anders als die Permutation, ist die Substitution noch nicht durch Nominal vordefiniert. Die Substitution (d.h. das Ersetzen) von x durch t in s wird als $s[x ::= t]$ geschrieben. Die Definition erfolgt rekursiv entlang der Definition von *lam*.

nominal_primrec *subst* :: *lam* $\Rightarrow$ *name* $\Rightarrow$ *lam* $\Rightarrow$ *lam* **where**
  $(V\ x)[y ::= s] = (\mathit{if}\ x = y\ \mathit{then}\ s\ \mathit{else}\ V\ x)$
| $(t\ \$\ t')[y ::= s] = t[y ::= s]\ \$\ t'[y ::= s]$
| $x \sharp (y, s) \Longrightarrow (\Lambda\ x.\ t)[y ::= s] = \Lambda\ x.(t[y ::= s])$

Die Vorbedingung im Abstraktions-Fall der Definition bewirkt, dass keine freien Variablen durch Substitution gebunden werden können. Diese Vorbedingung ist auch der Grund, warum hier **nominal_primrec** geschrieben werden muss, nicht **fun**.

Nun zu einigen Eigenschaften der Substitution: Ist x keine freie Variable von s, und ist ($x = y$ oder x keine freie Variable von t), dann kommt x nicht in den freien Variablen von t $[y ::= s]$ vor:

lemma *subst_fresh*:
  assumes $x \sharp s$ **and** $x = y \vee x \sharp t$
  shows $x \sharp t[y ::= s]$

Der Beweis dieses Lemmas funktioniert durch Induktion entlang von t, genauso wie der Beweis des nächsten Lemmas, welches besagt, dass für ein x, welches nicht frei in s vorkommt, die Substitution von x in s nichts verändert.

lemma *forget*:
  assumes $x \sharp s$
  shows $s[x ::= t] = s$

Mit diesen zwei technischen Lemmata kann das (ebenfalls technische) Substitutionslemma gezeigt werden. Das Substitutionslemma gibt unter gewissen Voraussetzungen eine Rechenregel zur Vertauschung von Substitutionen an:

lemma *substitution_lemma*:
  assumes $x \neq y$ **and** $x \sharp L$
  shows $M[x::=N][y::=L] = M[y::=L][x::=N[y::=L]]$

Der Beweis funktioniert durch Induktion entlang der Term-Struktur von M.

Eine weitere nützliche Eigenschaft der Substitution ist die Verträglichkeit mit Permutationen. Diese Verträglickeit nennt man Äquivarianz. Wendet man eine Permutation auf einen

Term an, in dem substituiert wurde, so kann man die Permutation auch erst auf den Term anwenden, auf die Variable, die ersetzt werden soll, und auf den einzusetzenden Term, und danach erst substituieren.

lemma *subst_equiv*:
  shows $pi \cdot (t[a ::= b]) = (pi \cdot t)[(pi \cdot a) ::= (pi \cdot b)]$

Der Beweis hierfür ist zweigeteilt. Einmal führt man Induktion entlang der Liste pi. Im Fall der einelementigen Liste führt man nochmals Induktion entlang der Termstruktur von t.

3.5 Auswertung durch "call-by-value"

In diesem Abschnitt werden die beiden Auswertungs-Relationen definiert, die später miteinander in Verbindung gebracht werden sollen.

Zunächst zur Auswertung der λ-Terme vom Typ *lam*. Diese Relation wird mit *cbv* bezeichnet und als Operator $\Downarrow$ geschrieben. Abstraktionen werden auf sich selbst reduziert, das besagt die erste Regel in der Definition. Die zweite Regel besagt im Prinzip, dass $\Lambda\ x.\ s\ \$\ t$ zu $s\ [x ::= t]$ ausgewertet wird. Würde man dies direkt hinschreiben, so hätte man eine Relation $p \Downarrow q$, die schrittweise vorgeht, und bei der ein Schritt von p zu q führt. (auf englisch nennt man sowas "small-step"-Semantik) Die Relation $p \Downarrow q$ soll aber hier bedeuten, dass p mit beliebig vielen Schritten auf q reduziert, wobei q selbst nicht mehr weiter reduziert werden kann. (auf englisch nennt man das "big-step"-Semantik) Also muss man die Definition so aufziehen: Angenommen $s\ [x ::= t]$ reduziert auf *res*, dann soll $\Lambda\ x.\ s\ \$\ t$ auch auf *res* reduzieren. Die call-by-value-Reduktion fordert, dass das Argument t erst eingesetzt wird, wenn es selbst schon vollständig reduziert wurde. Die erneut abgewandelte Definition sieht so aus: Angenommen $s\ [x ::= t']$ reduziert auf *res*, dann soll $\Lambda\ x.\ s\ \$\ t$ auch auf *res* reduzieren, falls $t \Downarrow t'$. Verallgemeinert man diesen Fall noch auf beliebige Applikationen $s\ \$\ t$, dann kommt man schließlich zur endgültigen Fassung der Definition:

inductive *cbv* :: *lam* $\Rightarrow$ *lam* $\Rightarrow$ *bool* **where**
  cbvL:
    $\Lambda\ x.\ t \Downarrow \Lambda\ x.\ t$
| *cbvA*:
    $[\![\ s \Downarrow \Lambda\ x.\ s'\ ;\ \ t \Downarrow t'\ ;\ \ s'[x ::= t'] \Downarrow res\ ]\!]$
      $\Longrightarrow s\ \$\ t \Downarrow res$

Der Abstraktion kommt dadurch die Rolle einer Funktion zu und der Applikation die Rolle der Funktionsanwendung. Man sieht auch, warum die Reduktion "call-by-value" heißt: Unter "Wert" (value) versteht man hier eine Abstraktionen. Es werden bei der Applikation erst Funktion und Argument auf Werte reduziert, bevor das Argument für den Platzhalter eingesetzt werden darf. (Erweitert man die call-by-value-Reduktion auf ganze Programmiersprachen, so kommen zu den Werten noch weitere Konstrukte hinzu, wie zum Beispiel ganze Zahlen.)

Dass die *cbv*-Reduktion auch immer zu Werten führt, besagt das folgende Lemma:

lemma *cbv_abs*:
  assumes $s \Downarrow t$
  shows $\exists\ x\ r.\ t = \Lambda\ x.\ r$

using *assms* **by** *induct auto*

Ausnahmsweise wurde hier einmal der Beweis dargestellt, um zu zeigen, was "einfach" in Isabelle heißt. Der Teil **using** *assms* besagt, dass die Annahmen verwendet werden sollen. **by** ist das Schlüsselwort, welchem die Beweismethode folgt. Danach besagt *induct*, dass Induktion geführt werden soll (die Art der Induktion ergibt sich aus der Annahme) und *auto* ist eine komplizierte Taktik in Isabelle, die versucht, den Rest des Beweises automatisch zu erledigen, was hier funktioniert.

Um den zweiten Auswertungsbegriff definieren zu können benötigt man einen weiteren Datentyp, den Typ der sog. Closures. Eine Closure (zu deutsch Abschluss) ist ein λ-Term zusammen mit einer Liste von Closures, die lang genug ist, damit alle deBruijn-Indizes, die im Term ungebunden sind, in die Liste hineinindizieren können. Die mitgeführte Liste nennt man auch Umgebung.

datatype *clos* = *Clos* (*clos list*) *db_lam*

Diese Closures werden bei der Auswertung verändert. In der folgenden Definition ist ! der Listen-Indizierungsoperator, sodass z.B. $[4, 5, 6] ! 1 = 5$ oder $[4, 5, 6] ! 0 = 4$. Diese Reduktion wird mit *Cbv* bezeichnet und als infix-Operator $\downarrow$ geschrieben. Die Definition ähnelt derjenigen von *cbv*, nur dass ein Fall für die Variablen hinzukommt, der zur Indizierung in die Umgebung dient, und dass an die Stelle der Substitution eine Erweiterung der Umgebung tritt.

```
inductive Cbv :: clos ⇒ clos ⇒ bool where
  cvL:
     Clos Γ (Abs t) ↓ Clos Γ (Abs t)
| cvV:
     ⟦ i < length Γ ;  Γ ! i ↓ res ⟧
        ⟹ Clos Γ (Var i) ↓ res
| cvA:
     ⟦ Clos Γ s ↓ Clos Δ (Abs s') ;  Clos Γ t ↓ t' ;
              Clos (t' # Δ) s' ↓ res          ⟧
           ⟹ Clos Γ (App s t) ↓ res
```

Die Liste (die Umgebung) übernimmt die Aufgabe einer Funktion, die Variablen Werte zuweist. Soll eine Variable reduziert werden, so muss zunächst der Wert aus der Liste reduziert werden. Anstatt dass bei Applikation substituiert wird, bekommt die Liste ein neues Element, was einer neuen Zuweisung eines Terms zur Variable mit Index 0 entspricht. (Sofern die Variable nicht unter Abstraktionen steht. Unter einer Abstraktion ist dann der Index 1 gemeint, unter zwei Abstraktionen der Index 2 usw.)

Abstraktionen bleiben weiterhin die einzigen Werte. Es lässt sich einfach zeigen, dass auch diese Reduktion immer Werte produziert, d.h. Abstraktionen:

```
lemma Cbv_abs:
  assumes s ↓ t
  shows ∃ N P. t = Clos N (Abs P)
using assms by induct auto
```

3.6 Übersetzung

3.6.1 Term-Übersetzung

Nun sollen die beiden Auswertungs-Begriffe zusammengeführt werden. Als erstes wird dazu eine Übersetzungsrelation namens *compile* definiert, welche die verschiedenen λ-Terme aufeinander abbildet. Zu dieser Übersetzung wird eine Liste von Variablen-Namen angesammelt, welche deBruijn-Indizes zu eben diesen Variablen-Namen übersetzt.

Die folgenden Regeln $c0$ und ci dienen dazu, deBruijn-Indizes in der Liste nachzuschlagen. Die Regel cL ist für das Übersetzen der Abstraktionen gedacht. In dieser Regel werden die abstrahierten Namen hinter dem Λ zur Liste hinzugefügt. Die letzte Regel cA besagt nur, dass Applikation komponentenweise übersetzt wird.

```
inductive compile :: name list ⇒ lam ⇒ db_lam ⇒ bool where
  c0:
     compile (x # xs) (V x) (Var 0)
| ci:
     ⟦ x ≠ y ;  compile xs (V x) (Var i) ⟧
    ⟹ compile (y # xs) (V x) (Var (Suc i))
| cL:
     ⟦ compile (x # xs) t t' ⟧
   ⟹ compile xs (Λ x. t) (Abs t')
| cA:
     ⟦ compile xs s s' ;  compile xs t t' ⟧
     ⟹ compile xs (s $ t) (App s' t')
```

Die Rechtseindeutigkeit muss erst bewiesen werden, d.h. es ist aus der Definition noch nicht ersichtlich, dass α-äquivalente (d.h. gleiche) *lam*-Terme auf eindeutige *db_lam*-Terme abgebildet werden.

Eine sehr schnell zu beweisende Tatsache ist: Falls *compile* $\Gamma\ t\ t'$ gilt, und *pi* eine Permutation ist, dann gilt auch *compile* $(pi \cdot \Gamma)\ (pi \cdot t)\ t'$. Dies ist die Äquivarianz der Relation *compile*.

```
lemma compile_equi:  assumes compile Γ t t'
  shows compile (pi · Γ) (pi · t) t'
using assms by induct (auto simp add:simps_perm)
```

Zum Beweis der Rechtseindeutigkeit der Übersetzung ist ersteinmal komplizierte Beweistechnik nötig. Als erstes werden zwei Funktions-Definitionen getätigt. Die erste Funktion heißt *name_chg*. Der Aufruf *name_chg x y n xs* soll n Vorkommen von x in xs überspringen, und danach alle Vorkommen von x in xs durch y ersetzen, und das Ergebnis dieser Ersetzung zurückliefern. Die Definition enthält entsprechend viele Fallunterscheidungen:

```
fun name_chg :: name ⇒ name ⇒ nat ⇒ name list ⇒ name list where
  nc0: name_chg x y 0 (z # xs) =
       (if z = x then y else z) # name_chg x y 0 xs
| ncS: name_chg x y (Suc n) (z # xs) =
       (z # name_chg x y (if z = x then n else Suc n) xs)
| ncN: name_chg x y n [] = []
```

Die zweite Funktion heißt *cover*, und ist eigentlich eine Relation, als Funktion hingeschrieben,

denn das Ergebnis hat den Typ *bool*. Die Relation *cover x y n xs* soll bedeuten, dass alle Vorkommen von x in xs von links gesehen durch ein Vorkommen von y verdeckt werden, bis auf eventuell n Vorkommen von x in xs. Taucht also in xs mehr als $n$mal x auf, so muss spätestens zwischen dem $n$ten und dem $n+1$ten x ein y stehen.

```
fun cover :: name ⇒ name ⇒ nat ⇒ name list ⇒ bool where
  cv0: cover x y 0 (z # xs) =
         (if z = y then True else
          if z = x then False
          else cover x y 0 xs)
| cvS: cover x y (Suc n) (z # xs) =
         (if z = y then True else
          if z = x then cover x y n xs
          else cover x y (Suc n) xs)
| cvN: cover x y n [] = True
```

Es gilt trivial:

```
lemma hd_cover:
  shows cover x y n (y # xs)
```

Das nächste Lemma folgt direkt aus der Definition:

```
lemma name_chg_neq:
  assumes x ≠ z
  shows name_chg x y n (z # xs) = z # name_chg x y n xs
```

Die beiden vorangegangenen Lemmata sind notwendig für den Beweis des folgenden Lemmas. Es setzt *compile xs s t* voraus und besagt, dass man unter gewissen Bedingungen folgern kann, dass *compile* (*name_chg x y n xs*) *s t* gilt. Die Bedingungen sind gerade so gewählt, dass sie zu den späteren Anwendungen passen. :)

```
lemma cover_name_chg:
  assumes compile xs s t and x ♯ s ∨ n > 0 and y ♯ s ∨ cover x y n xs
  shows compile (name_chg x y n xs) s t
```

Der Beweis läuft mit Induktion fast automatisch durch. Interessanter als der Beweis ist die Tatsache, dass man mit diesem Lemma zur Eindeutigkeit von *compile* kommen kann.

Um das Lemma *cover_name_chg* anwenden zu können, muss man es erstmal mit Permutationen in Verbindung bringen. Die Anwendungsfälle beinhalten entweder $n = 0$ oder $n = 1$, man kann also die Definition von *cover* direkt auflösen. Die Funktion *cover* kommt im folgenden nicht mehr vor, sie wurde nur für den Beweis von *cover_name_chg* benötigt.

Um *name_chg* kümmern sich die folgenden drei Lemmata.

```
lemma name_chg_swap0:
  assumes z ♯ xs
  shows name_chg x z 0 xs = [(z, x)] ∙ xs

lemma name_chg_swap1:
  assumes z ♯ xs
```

```
  shows name_chg x z 1 (x # xs) = x # [(z, x)] · xs
using assms name_chg_swap0 by auto

lemma name_chg_swap2:
  assumes z ♯ xs and y ≠ z
  shows name_chg y z 0 (z # xs) = z # [(z, y)] · xs
using assms name_chg_swap0 by auto
```

Nun zum ersten Lemma, dessen Beweis in dieser Arbeit etwas detailierter erklärt wird. Vorausgesetzt wird, dass y nicht frei in s vorkommt, sowie *compile* $(x \# xs)\ s\ t$, wobei $x \neq y$. Dann kann man anstatt mit x auch mit y vor xs den Term s nach t übersetzen, wenn man nur x und y in s vertauscht. Dies ergibt sich nicht durch *compile_equi*. Das Ergebnis für *compile_equi* würde lauten: *compile* $(y \# [(x, y)] \cdot xs)\ ([(x, y)] \cdot s)\ t$. Die Elimination der Permutation $[(x, y)]$ in der Liste ist das Besondere am folgenden Lemma:

```
lemma compile_swap':
  assumes y ♯ s and compile (x # xs) s t and x ≠ y
  shows compile (y # xs) ([(x, y)] · s) t
proof −
```

Zunächst wählt man ein z, welches noch nirgends auftritt:

```
  obtain z where  z ♯ (xs, x, y, s)
```

Aus *cover_name_chg* mit $n = 1$ folgt:

compile $(name\_chg\ x\ z\ 1\ (x \# xs))\ s\ t$

Zusammen mit *name_chg_swap*1 ergibt sich:

compile $(x \# [(z, x)] \cdot xs)\ s\ t$

Daraus kann man mit *cover_name_change* für $n = 0$ folgern:

compile $(name\_chg\ y\ x\ 0\ (x \# [(z, x)] \cdot xs))\ s\ t$

Dabei ist die *cover*-Bedingung trivial erfüllt. Mit *name_chg_swap*2 wird daraus:

compile $(x \# [(x, y), (z, x)] \cdot xs)\ s\ t$

Eine dritte Anwendung von *cover_name_chg* liefert für $n = 0$:

compile $(name\_chg\ z\ y\ 0\ (x \# [(x, y), (z, x)] \cdot xs))\ s\ t$

Daraus wird schließlich mit name_chg_swap2:

compile $(x \# [(y, z), (x, y), (z, x)] \cdot xs)\ s\ t$

Wie man leicht nachrechnet, vereinfacht sich die Permutation zu $[(x, y)]$:

compile $(x \# [(x, y)] \cdot xs)\ s\ t$

Wendet man darauf nun das Lemma *compile_equi* für $pi = [(x, y)]$ an, so ist der Beweis für *compile_swap'* komplett.

compile $(y \# xs)\ ([(x, y)] \cdot s)\ t$
qed

Die Bedingung $x \neq y$ aus den Voraussetzungen von *compile_swap'* kann weggelassen werden, da für $x = y$ garnichts zu zeigen ist:

lemma *compile_swap*:
  assumes $y \sharp s$ **and** *compile* $(x \# xs)$ s t
  shows *compile* $(y \# xs)$ $([(x, y)] \cdot s)$ t

Als direkte Folgerung erhält man eine spezielle Umkehrung der Regel *cL* aus der Definition von *compile*. Das *inductive*-Paket von Isabelle generiert automatisch eine allgemeine Umkehrung der Definition und damit auch von *cL*: Vorausgesetzt man hat *compile xs* $(\Lambda\ x.\ s)$ $(Abs\ t)$ gezeigt, dann kann man daraus folgern, dass *compile* $(z \# xs)$ r t für ein z und ein r mit $\Lambda\ x.\ s = \Lambda\ z.\ r$ gilt, weil intuitiv die Regel *cL* die einzige Regel ist, mit der man den Beweis hätte führen können. Da die λ-Terme α-Äquivalenz-Klassen sind, muss hier nicht unbedingt $x = z$ gelten. Dass *compile* $(x \# xs)$ s t dennoch gilt, besagt die speziellere Umkehrung:

lemma *compile_abs_inv*:
  assumes *compile xs* $(\Lambda\ x.\ s)$ $(Abs\ t)$
  shows *compile* $(x \# xs)$ s t
proof −

Zunächst erhält man z und r durch die autmatisch generierte Umkehrung:

  obtain $z\ r$ **where**
    zr1: *compile* $(z \# xs)$ r t **and**
    zr2: $\Lambda\ z.\ r = \Lambda\ x.\ s$

Für $z = x$ ist nichts mehr zu zeigen. Also kann man $z \neq x$ annehmen, und $\Lambda\ z.\ r = \Lambda\ x.\ s$ ausnutzen:

    assume *asm1*: $z \neq x$ **and** *asm2*: $s = [(z, x)] \cdot r$ **and** *asm3*: $x \sharp r$

Das reicht schon für die Anwendung von *compile_swap* auf *zr*1:

    compile $(x \# xs)$ $([(z, x)] \cdot r)$ t

Mit *asm*2 wird $[(z, x)] \cdot r$ zu s:

  compile $(x \# xs)$ s t
    qed

Nun zur lange angekündigten Eindeutigkeit von *compile*. Im Beweis wird nur der interessante Fall der Induktion gezeigt.

lemma *compile_unique*:
  assumes *compile xs s t* **and** *compile xs s* t'
  shows $t = t'$
proof (*induct t arbitrary*:t')

Die Eindeutigkeit wird durch Induktion entlang der Definition von *compile* geführt.

Der interessante Fall: Die Regel *cL*.

  case *cL*

Die Induktions-Voraussetzung lautet:
$\forall t'.\ compile\ (x\ \#\ xs)\ s\ t' \longrightarrow t = t'$
Vorausgesetzt werden $compile\ xs\ (\Lambda\ x.\ s)\ (Abs\ r)$ und $compile\ xs\ (\Lambda\ x.\ s)\ (Abs\ t)$.
Zu zeigen ist $t = r$.

Aus den Annahmen folgt mit *compile_abs_inv*:

$compile\ (x\ \#\ xs)\ s\ r$

Nach Induktions-Voraussetzung folgt:

$t = r$

Und das war's schon.
Dieser Schluss hätte nicht funktioniert, wenn man nur $compile\ (z\ \#\ xs)\ s'\ r$ mit $\Lambda z.s' = \Lambda x.s$ durch die Inversion erhalten hätte. Die spezielle Umkehrung ist nötig. In den restlichen Induktions-Fällen ergibt sich die Behauptung durch einfache Anwendungen der Induktionsvoraussetzungen, ohne Komplikationen.

qed

3.6.2 Closure-Übersetzung

Nachdem Terme nun übersetzt werden können, sind die Closures dran. Da Closures Listen enthalten, und die Übersetzung mit Substitution arbeiten wird, ist es ersteinmal notwendig, Substitution auf Listen zu definieren.

```
fun substs :: (name × lam) list ⇒ lam ⇒ lam where
  substs [] t = t
| substs ((x, s) # Γ) t = substs Γ (t[x ::= s])
```

Viele Lemmata über *subst* lassen sich auf *substs* übertragen. Zum Beispiel wird *subst_fresh* zu folgendem *substs_freshs*. Die Funktion *set* verwandelt dabei eine Liste in eine Menge.

```
lemma substs_freshs:
  assumes x ♯ t ∨ x ∈ set (map fst Γ)
  assumes ∀ y r. (y, r) ∈ set Γ ⟶ x ♯ r
  shows x ♯ substs Γ t
```

Jetzt sollen Closures und Terme vom Typ *lam* ineinander übersetzt werden. Dazu müssen alle Elemente der Liste der Closure eingesetzt werden. Die Übersetzung muss bereits in der Definition auf Listen von Closures erweitert werden.

```
inductive compile_clos :: lam ⇒ clos ⇒ bool
  and compile_closs :: lam list ⇒ clos list ⇒ bool where
  cv'0:
     ⟦ compile (map fst Γ) t t' ;  compile_closs (map snd Γ) M ⟧
          ⟹ compile_clos (substs Γ t) (Clos M t')
| cvs0:
     compile_closs [] []
| cvsC:
     ⟦ compile_clos (Λ x. s) (Clos M (Abs t)) ;  compile_closs Γ Γ' ⟧
       ⟹ compile_closs ((Λ x. s) # Γ) ((Clos M (Abs t)) # Γ')
```

Die Übersetzung auf Listen erfordert hier, dass die Liste nur Abstraktionen enthält. Dies vereinfacht die späteren Beweise.

Eine wichtige Erkenntnis über das Kompilieren von Closures ist, dass das das Kompilieren eines deBruijn-Index äquivalent ist zum Kompilieren des entsprechenden Terms aus der Umgebung der Closure. Auf diese Weise kann der Variablen-Fall in Induktionen mit *compile_clos* auf den Abstratktions-Fall zurückgeführt werden.

lemma *compile_clos_Var*:
  assumes *compile_clos s* (*Clos M* (*Var i*))
  shows *compile_clos s* (*M* ! *i*)
proof −

Nach Definition von *compile_clos* darf man Γ und t voraussetzen mit $s = substs\ \Gamma\ t$ und den folgenden Eigenschaften:

  assume *asm*: *compile* (*map fst* Γ) *t* (*Var i*)
   and *asm'*: *compile_closs* (*map snd* Γ) *M*

Zu zeigen ist dann *compile_clos* (*substs* Γ *t*) (*M* ! *i*).

Aus den Annahmen kann man schließen, dass t eine Variable Vv ist, dass in Γ an $i$ter Stelle v steht, und alle Variablen davor in Γ von v verschieden sind:

  from *asm* **obtain** *v* **where**
    v: *t* = *V v*
    and *map fst* Γ ! *i* = *v*
    and ∀ *j* < *i*. *map fst* Γ ! *j* ≠ *v*
    and *vi*: *i* < *length* (*map fst* Γ)

An dieser Stelle wird ein wenig technisches Beweisen über *compile* weggelassen. Es lässt sich nun folgern, dass Substitution von Γ in t sich auf Indizierung in Γ reduziert:

  eq: *substs* Γ (*V v*) = *snd* (Γ ! *i*)

Aus asm' lässt sich folgern:

  compile_clos ((*map snd* Γ) ! *i*) (*M* ! *i*)

denn *compile_closs* ist nur die auf Listen erweiterte Version von *compile_clos*. Mit *eq* und $t = Vv$ folgt daraus die Behauptung:

  compile_clos (*substs* Γ *t*) (*M* ! *i*)

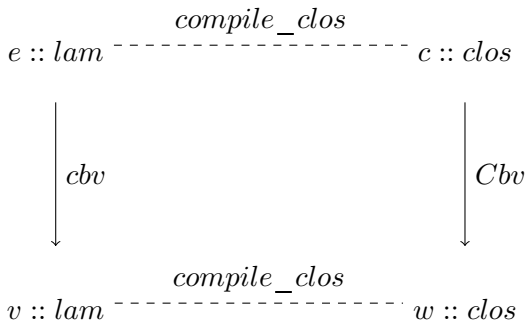
qed

Wie bei *compile* kann man auch bei *compile_clos* Permutationen anwenden, ohne dass sich das Kompilat verändert:

lemma *compile_clos_equi*:
  assumes *compile_clos t t'*
 shows *compile_clos* (*pi* · *t*) *t'*

4 Verifikation der Übersetzung

In diesem Abschnitt sollen die Beweise geführt werden, deren Ergebnisse in folgendem Diagramm zusammengefasst sind:



Der Term e entspricht dem "Quelltext", die Closure c dem Kompilat. Der Wert v ist die Auswertung von e und w die Auswertung von c. Die Verbindung zwischen e und c gehört immer zur Voraussetzung. Das bedeutet es wird immer von einer Closure und einem Term ausgegangen, die sich ineinander übersetzen lassen. Im Vollständigkeits-Lemma wird w und die Verbindung zwischen c und w vorausgesetzt. Zu zeigen ist dann, dass ein v existiert, mit den entsprechenden fehlenden Verbindungen. Im Korrektheits-Lemma wird v mit der Verbindung zu e vorausgesetzt, und es ist die Existenz von w zu zeigen mit den fehlenden Verbindungen. Isgesamt könnte man daraus folgern, dass e sich genau dann auf ein v reduzieren lässt, wenn c sich auf ein w reduzieren lässt, und falls dies der Fall ist, dass sich dann v und w wieder ineinander übersetzen lassen. Diese zusammengefasste Version wird hier nicht bewiesen, sondern nur die dazu notwendigen Lemmata für die Korrektheit und für die Vollständigkeit.

4.1 Vollständigkeit

Nun zum vorvorletzten Lemma dieser Arbeit. Es handelt sich um die Verträglichkeit von *compile_clos* mit *cL* bzw. *cbvA*, d.h. mit den Entscheidenden Schritten der beiden Auswertungsrelationen. Dieses Lemma wird beim Beweis der "Vollständigkeit" weiter unten benötigt.

Angenommen wird, dass $\Lambda\ x.\ s$ zu $Clos\ \Gamma\ (Abs\ s')$ kompiliert und dass t zu $Clos\ \Delta\ (Abs\ t')$ kompiliert. Ein paar Erklärungen bevor die Konklusion behauptet wird: Mit der Auswertungsrelation *cbv* reduziert $\Lambda\ x.\ s\ \$\ t$ auf das gleiche Ergebnis, auf das $s[x ::= t]$ reduziert. Mit *Cbv* reduziert $Clos\ \Delta\ (Abs\ t')$ auf sich selbst, und $Clos\ \Gamma\ (App\ s'\ t'')$ auf das gleiche Ergebnis, das man aus $Clos\ (Clos\ \Delta\ (Abs\ t')\ \#\ \Gamma)\ s'$ erhält, vorausgesetzt, dass $Clos\ \Gamma$

t'' auf *Clos* Δ (*Abs* t') reduziert. Durch diese Reduktionen von $\Lambda\ x.\ s\ \$\ t$ zu $s\ [x ::= t]$ und von *Clos* Γ (*App* $s'\ t''$) zu *Clos* (*Clos* Δ (*Abs* t') # Γ) s' erklärt sich die Konklusion, die behauptet, dass eben diese beiden um einen Schritt reduzierten Terme wieder ineinander übersetzt werden können: *compile_clos* ($s\ [x ::= t]$) (*Clos* (*Clos* Δ (*Abs* t') # Γ) s').

Der Beweis benötigt keine Induktion:

lemma *compile_subst*:
  assumes *a1*: *compile_clos* ($\Lambda\ x.\ s$) (*Clos* Γ (*Abs* s'))
  assumes *a2*: *compile_clos* t (*Clos* Δ (*Abs* t'))
  shows *compile_clos* ($s[x ::= t]$) (*Clos* (*Clos* Δ (*Abs* t') # Γ) s')
proof −

Als erstes wird die Definition von *compile_clos* in *a*1 aufgefaltet. Man erhält ein Θ und ein *s*0 mit den folgenden Eigenschaften:

  obtain Θ *s0* **where**
    Θs0: *compile* (*map fst* Θ) *s0* (*Abs* s')
      $\wedge\ \Lambda\ x.\ s$ = *substs* Θ *s0*
      $\wedge$ *compile_closs* (*map snd* Θ) Γ

Da *s*0 auf eine Abstraktion kompiliert, muss *s*0 ebenfalls eine Abstraktion sein:

  then obtain $y\ r$ **where** *yr*: *s0* = $\Lambda\ y.\ r$

Nun hat man alle Variablen, die man benötigt, um ein z wählen zu können, das frisch genug ist:

  then obtain z :: *name* **where** *z*: $z\ \sharp\ (\Theta, r, x, s)$

Mit diesem z kann man die gebundene Variable y in *s*0 umbenennen:

  s0: *s0* = ($\Lambda\ z.\ ([(y, z)] \cdot r)$)

Aus *Θs0* folgt auch eine alternative Darstellung von $\Lambda\ x.\ s$:

  $\Lambda\ x.\ s = \Lambda\ z.$ *substs* Θ ($[(y, z)] \cdot r$)

In dieser Gleichung kann man die Abstraktion weglassen, wenn man auf der rechten Seite z mit x vertauscht.

  s: $s = [(x, z)] \cdot$ *substs* Θ ($[(y, z)] \cdot r$)

Auf die Gleichung *s* wird am Schluss des Beweises zurückgegriffen.

Aus *Θs0* folgt mit *compile_abs_inv*:

  c0: *compile* (*map fst* ((z, t) # Θ)) ($[(y, z)] \cdot r$) s'

Ebenfalls aus *Θs0* und den Annahmen folgt:

  c1: *compile_closs* (*map snd* ((z, t) # Θ)) (*Clos* Δ (*Abs* t') # Γ)

Daraus bekommt man mit der Definition von *compile_clos*:

  compile_clos
    (*substs* ((z, t) # Θ) ($[(y, z)] \cdot r$))

$$(Clos\ (Clos\ \Delta\ (Abs\ t')\ \#\ \Gamma)\ s')$$

Mit *compile_clos_equi* für die Permutation $[(x, z)]$ erhält man:

$$\begin{aligned}&*\text{: } compile\_clos\\ &\quad([(x,\ z)] \cdot (substs\ ((z,\ t)\ \#\ \Theta)\ ([(y,\ z)] \cdot r)))\\ &\quad(Clos\ (Clos\ \Delta\ (Abs\ t')\ \#\ \Gamma)\ s')\end{aligned}$$

Das Kompilat, d.h. der Term vom Typ *db_lam* stimmt schon mit der Konklusion des Lemmas überein. Nun muss nur noch das erste Argument vom Typ *lam* auf die passende Form gebracht werden. Als erstes wird die Definition von *substs* bemüht:

$$\begin{aligned}&[(x,\ z)] \cdot (substs\ ((z,\ t)\ \#\ \Theta)\ ([(y,\ z)] \cdot r))\\ &= [(x,\ z)] \cdot (substs\ \Theta\ (([(y,\ z)] \cdot r)[z ::= t]))\end{aligned}$$

Jetzt hilft die Äquivarianz der Substitution weiter:

$$\ldots = ([(x,\ z)] \cdot substs\ \Theta\ ([(y,\ z)] \cdot r))[x ::= ([(x,\ z)] \cdot t)]$$

x und z kommen in t nicht mehr vor. Desweiteren lässt sich jetzt die Gleichung s benutzen, der Ausdruck vereinfacht sich radikal zu:

$$\ldots = s[x ::= t]$$

Aus dieser Gleichungskette folgt mit $*$ die Behauptung.

qed

Das nächste Lemma ist die Vollständigkeit. Sie ermöglicht es, von der Ausführung von *db_lam*-Termen auf die Ausführung von *lam*-Termen zu schließen. Die Closure c aus dem Diagramm oben sei nun $Clos\ \Gamma\ t$. Die Annahmen sind $Clos\ \Gamma\ t \downarrow w$ (die Verbindung rechts im Diagramm) und $compile\_clos\ e\ (Clos\ \Gamma\ t)$ (die Verbindung oben im Diagramm). Daraus kann man dann folgern, dass e auf ein v reduziert, welches in w übersetzt werden kann:

lemma *completeness*:
  assumes $Clos\ \Gamma\ t \downarrow w$
  assumes $compile\_clos\ e\ (Clos\ \Gamma\ t)$
  shows $\exists\ v.\ e \Downarrow v \wedge compile\_clos\ v\ w$
proof (*induct arbitrary*:*e rule*:*Cbv.induct*)

Der Beweis benutzt Induktion entlang der Definition von $\_ \downarrow \_$. Der Variablen-Fall ist trivial, wenn man $compile\_clos\_Var$ benutzt, denn er reduziert sich auf den Abstraktions-Fall.

  next

Der zweite Fall ist der Abstraktions-Fall. Dieser Fall ist ebenfalls trivial, denn es ist e ein Wert. Es wird $v = e$ gewählt:

  case *cvL*
  obtain $x\ s$ **where** $e = \Lambda\ x.\ s$

Zu zeigen bleibt:

$e \Downarrow e$

was nach Definition gilt.

next

Als letztes der interessante Fall:

case *cvA*

In diesem Fall ist e eine Applikation:

obtain e_1 e_2 **where** e: $e = e_1$ \$ e_2

Die beiden Komponenten der Applikation können getrennt auf die Komponenten von $t = App\ t_1\ t_2$ übersetzt werden nach Voraussetzung:

e': *compile_clos* e_1 (*Clos* Γ t_1) **and** *compile_clos* e_2 (*Clos* Γ t_2)

Nach Induktionsvoraussetzung reduziert e_1 auf einen Wert, der in eine Abstraktion uebersetzt werden kann, ebenso e_2:

obtain v_1 v_2 **where**
vs0: $e_1 \Downarrow v_1$
and *vs1*: *compile_clos* v_1 (*Clos* Δ (*Abs* s_1))
and *vt0*: $e_2 \Downarrow v_2$
and *vt1*: *compile_clos* v_2 s_2

Die Induktionsvoraussetzung ergibt auch:

ih: $\forall\ p.$ *compile_clos* p (*Clos* (s_2 # Δ) s_1) $\longrightarrow$ ($\exists\, v.\ p \Downarrow v \wedge$ *compile_clos* $v\ w$)

Gesucht wird ein v mit $e \Downarrow v$ und *compile_clos* $v\ w$. Dazu soll *ih* benutzt werden, wobei das p noch so zusammengebastelt werden muss, dass die Prämisse von *ih* gilt.

Die konkrete Darstellung der Abstraktion v_1 (es ist eine wegen *cbv_abs*) sei die folgende:

obtain $y\ r$ **where** *yr*: $v_1 = \Lambda\ y.\ r$

Die Darstellung der Closure s_2 (der Term in der Closure ist eine Abstraktion nach *Cbv_abs*) sei:

obtain $\Delta'\ r'$ **where** $s_2 =$ *Clos* Δ' (*Abs* r')

Hieraus, aus *yr*, *vs*1 und *vt*1 folgt mit *compile_subst*:

compile_clos ($r[y ::= v_2]$) (*Clos* (s_2 # Δ) s_1)

Das gesuchte p für *ih* ist nun $r[y ::= v_2]$. Aus *ih* erhält man mit eben gezeigtem:

obtain v **where**
$r[y ::= v_2] \Downarrow v$ **and** v: *compile_clos* $v\ w$

Mit *vs*0 und *vt*0 in eventuell anderen Darstellungen und mit eben bewiesenem folgt nach Definition der Relation _ $\Downarrow$ _ der nach v noch fehlende Teil der Induktions-Behauptung:

$e \Downarrow v$

qed

4.2 Korrektheit

Als letztes wird die Korrektheit bewiesen. Angenommen es gilt $e \Downarrow v$ und e kann zu $Clos\ \Gamma\ t$ übersetzt werden. Dann wird auch $Clos\ \Gamma\ t$ zu einem w reduziert, welches zu v zurückübersetzt werden kann.

lemma *soundness*:
  assumes $e \Downarrow v$
  assumes *compile_clos* e $(Clos\ \Gamma\ t)$
  shows $\exists\ w.\ Clos\ \Gamma\ t \downarrow w \wedge$ *compile_clos* $v\ w$
proof (*induct arbitrary*: $\Gamma\ t$ *rule*:*cbv.induct*)

Der Beweis wird mit Induktion entlang der Definition von _ ⇓ _ geführt. Als erstes der einfachere Fall, der Abstraktionsfall:

  case *cbvL*

In diesem Fall ist e eine Abstraktion, etwa $\Lambda\ x.\ e_1$. Daraus folgt, dass t entweder eine Variable ist, oder eine Abstraktion:

  $(\exists\ i.\ t = Var\ i) \vee (\exists\ t_1.\ t = Abs\ t_1)$

Angenommen, t ist eine Variable:

  assume $t = Var\ i$

Dann erhält man nach Definition von *compile_clos* aus den Annahmen ein Θ und ein e' mit den folgenden Eigenschaften:

  obtain $\Theta\ e'$ **where**
    Gt': *compile* $(map\ fst\ \Theta)\ e'\ t \wedge$
      compile_closs $(map\ snd\ \Theta)\ \Gamma \wedge$
      substs $\Theta\ e' = \Lambda\ x.\ e_1$

Desweiteren muss die Closure in der Umgebung Γ, welche dem Index i entspricht, ebenfalls eine Abstraktion sein:

    obtain $\Delta\ t_1$ **where** *Dt*: $\Gamma\ !\ i = Clos\ \Delta\ (Abs\ t_1)$

Mit *compile_clos_Var* ist nun klar, wie $\Lambda x.e_1$ übersetzt wird:

  A: *compile_clos* $(\Lambda\ x.\ e_1)$ $(Clos\ \Delta\ (Abs\ t_1))$

Von der Abstraktion in Γ ist sowieso klar, worauf sie reduziert, nämlich auf sich selbst:

  B: $\Gamma\ !\ i \downarrow Clos\ \Delta\ (Abs\ t_1)$

Mit *A* weiß man deshalb, worauf $Clos\ \Gamma\ t$ reduziert:

  $Clos\ \Gamma\ (Var\ i) \downarrow Clos\ \Delta\ (Abs\ t_1)$

Der Fall $t = Var\ i$ ist also erledigt, wenn man $w = Clos\ \Delta\ (Abs\ t_1)$ wählt.

Der Abstraktions-Fall ist trivial, gewählt wird $w = Clos\ \Gamma\ t$:

assume $t = Abs\ t_1$

Wie $Clos\ \Gamma\ t$ reduziert ist klar:

$Clos\ \Gamma\ t \downarrow Clos\ \Gamma\ t$

Aus den Annahmen geht direkt die zweite für w zu zeigende Eigenschaft hervor:

$compile\_clos\ (\Lambda\ x.\ e_1)\ (Clos\ \Gamma\ t)$

Der Abstraktionsfall der Induktion ist damit fertig.

next

Nun der interessantere Fall der Induktion:

case *cbvA*

In diesem Fall ist e eine Applikation, $e = e_1\ \$\ e_2$ Vorausgesetzt werden $e_1 \Downarrow \Lambda\ x.\ s_1$, $e_2 \Downarrow s_2$ und $s_1\ [x ::= s_2] \Downarrow v$. Für jede dieser Voraussetzungen der Form $p \Downarrow q$ gilt die Induktionsvoraussetzung: $\forall\, cl.\ compile\_clos\ p\ cl \longrightarrow (\exists\, w'.\ cl \downarrow w' \wedge compile\_clos\ q\ w')$. Eine weitere Annahme ist $compile\_clos\ (e_1\ \$\ e_2)\ (Clos\ \Gamma\ t)$. Diese Voraussetzungen müssen jetzt so ausgenutzt werden, dass man ein w erhält, mit $Clos\ \Gamma\ t \downarrow w \wedge compile\_clos\ v\ w$.

Zunächst folgt, dass die Übersetzung t von $e_1\ \$\ e_2$ ebenfalls eine Applikation sein muss:

obtain $t_1\ t_2$ **where** *t1t2*: $t = App\ t_1\ t_2$

Die beiden Komponenten der Applikation können jetzt auch getrennt übersetzt werden:

s: $compile\_clos\ e_1\ (Clos\ \Gamma\ t_1)$
t: $compile\_clos\ e_2\ (Clos\ \Gamma\ t_2)$

Nun kann man die Induktionsvoraussetzung für e_1 anwenden, wobei $cl = Clos\ \Gamma\ t_1$ gewählt wird. Man erhält ein w_1' mit folgenden Eigenschaften:

obtain $w_1{}'$ **where** *W*: $Clos\ \Gamma\ t_1 \downarrow w_1{}' \wedge compile\_clos\ (\Lambda\ x.\ s_1)\ w_1{}'$

Daraus folgt, dass w_1' eine Closure einer Abstraktion sein muss:

obtain $\Delta_1\ w_1$ **where** *NP*: $w_1{}' = (Clos\ \Delta_1\ (Abs\ w_1))$

Faltet man die Definition von *compile_clos* in W zusammen mit NP auf, so erhält man ein Θ und ein r' mit den Voraussetzungen der Übersetzung:

obtain $\Theta\ r'$ **where**
Gr0: $compile\ (map\ fst\ \Theta)\ r'\ (Abs\ w_1)$
and *Gr1*: $compile\_closs\ (map\ snd\ \Theta)\ \Delta_1$
and *Gr2*: $(\Lambda\ x.\ s_1) = substs\ \Theta\ r'$

Daraus folgt, dass r' eine Abstraktion sein muss. Dies ermöglicht ein weiteres Auffalten einer Definition, diesmal von *compile*.

obtain $y\ r$ **where**
yr0: $r' = \Lambda\ y.\ r$

and *yr1*: *compile* $(y \# map\ fst\ \Theta)\ r\ w_1$

Das gleiche Spiel wird nocheinmal mit der Induktionsvoraussetzung für e_2 wiederholt. Man erhält ein w_2' mit folgenden Eigenschaften:

obtain $w_2{}'$ **where**
t2w2: *Clos* $\Gamma\ t_2 \downarrow w_2{}'$
and *s2w2*: *compile_clos* $s_2\ w_2{}'$

w_2' muss natürlich ein Wert sein:

obtain $\Delta_2\ w_2$ **where** $\Gamma' Q$: $w_2{}' = Clos\ \Delta_2\ (Abs\ w_2)$

Aus $Gr1$ und $s2w2$ folgt:

compile_closs $(s_2 \# map\ snd\ \Theta)\ (w_2{}' \# \Delta_1)$

Ein wenig umgeschrieben sieht das so aus:

compile_closs $(map\ snd\ ((y,\ s_2) \# \Theta))\ (w_2{}' \# \Delta_1)$

$yr1$ kann man ebenfalls etwas umschreiben:

compile $(map\ fst\ ((y,\ s_2) \# \Theta))\ r\ w_1$

Aus den letzten beiden Tatsachen folgt nach Definition:

compile_clos $(substs\ ((y,\ s_2) \# \Theta)\ r)\ (Clos\ (w_2{}' \# \Delta_1)\ w_1)$

Nach Definition von *substs* ergibt sich:

compv: *compile_clos* $(substs\ \Theta\ (r\ [y ::= s_2]))\ (Clos\ (w_2{}' \# \Delta_1)\ w_1)$

Das erste Argument von *compile_clos* muss nun umgeschrieben werden, um ein drittes mal die Induktionsvoraussetzung, diesmal für $s_1[x ::= s_2]$ anwenden zu können. Dazu wird ein z benötigt, welches nirgends frei vorkommt:

obtain z :: *name* **where** z: $z \sharp (\Theta,\ r,\ y,\ s_1,\ x,\ s_2)$

Mit diesem z wird $\Lambda x.s_1$ umgeschrieben:

xs0: $\Lambda\ x.\ s_1 = \Lambda\ z.\ ([(z,\ x)] \cdot s_1)$

Auch in $\Lambda y.r$ wird die gebundene Variable umbenannt:

yr: $\Lambda\ y.\ r = \Lambda\ z.\ ([(z,\ y)] \cdot r)$

Aus $Gr2$ und $yr0$ erhält man:

substs $\Theta\ (\Lambda\ y.\ r) = \Lambda\ x.\ s_1$

Setzt man die umgeschriebenen Versionen von $\Lambda y.r$ und $\Lambda x.s_1$ ein, folgt:

substs $\Theta\ (\Lambda\ z.\ ([(z,\ y)] \cdot r)) = \Lambda\ z.\ ([(z,\ x)] \cdot s_1)$

Da das z nach Wahl nicht in Θ auftreten kann, ist es möglich, die Abstraktion über *substs* hinwegzuheben:

$\Lambda\ z.\ (substs\ \Theta\ ([(z,\ y)] \cdot r)) = \Lambda\ z.\ ([(z,\ x)] \cdot s_1)$

Da links und rechts die gebundenen Variablen gleich sind, kann man die Abstraktion weglassen:

$substs\ \Theta\ ([(z,\ y)] \cdot r) = [(z,\ x)] \cdot s_1$

Auf beiden Seiten wird nun z durch s_2 ersetzt:

$(substs\ \Theta\ ([(z,\ y)] \cdot r))[z ::= s_2] = ([(z,\ x)] \cdot s_1)[z ::= s_2]$

Mit der Äquivarianz der Substitution kann man Substitution und Permutation auf der rechten Seite zusammenfassen:

$(substs\ \Theta\ ([(z,\ y)] \cdot r))[z ::= s_2] = (s_1\ [x ::= s_2])$

Auf der linken Seite kann man nach Wahl von z die Substitution $z ::= s_2$ unter das *substs* ziehen:

$substs\ \Theta\ (([(z,\ y)] \cdot r)[z ::= s_2]) = s_1\ [x ::= s_2]$

Wieder kann man mit der Äquivarianz die Permutation und die Substitution zusammenfassen:

substs_eq: $substs\ \Theta\ (r\ [y ::= s_2]) = s_1\ [x ::= s_2]$

Dies ist genau die Gleichung, mit der jetzt *compv* umgeschrieben werden soll. Das Ergebnis ist:

$compile\_clos\ (s_1[x ::= s_2])\ (Clos\ (w_2{}'\ \#\ \Delta_1)\ w_1)$

Benutzt man dies als Prämisse für die Induktionsvoraussetzung, so erhält man schließlich das gesuchte w:

obtain w **where** *res0*: $Clos\ (w_2{}'\ \#\ \Delta_1)\ w_1 \downarrow w$ **and** *res1*: $compile\_clos\ v\ w$

Einzig zu zeigen ist noch, dass $Clos\ \Gamma\ t$ auf w reduziert. Dies ergibt sich durch Anwendung der Definition von $\_ \downarrow \_$. Aus W und NP folgt nämlich:

$Clos\ \Gamma\ t_1 \downarrow Clos\ \Delta_1\ (Abs\ w_1)$

Aus $t2w2$ folgt zusätzlich:

$Clos\ \Gamma\ t_2 \downarrow w_2{}'$

Damit hat man:

$Clos\ \Gamma\ (App\ t_1\ t_2) \downarrow w$

was mit $t1t2$ der Behauptung entspricht.

qed

5 Ausblick

In dieser Arbeit wurden zwei Varianten des λ-Kalkül definiert, ineinander übersetzt, und die Korrektheit der Übersetzung wurde bewiesen. Diese Formalisierung könnte in Zukunft benutzt werden, um einen verifizierten Compiler für eine komplette funktionale Programmiersprache zu schreiben. Dazu notwendig wäre eine Erweiterung der Datentypen *lam* und *db_lam*, entsprechende Erweiterungen der Auswertungs-Relationen, und eine Anpassung der Übersetzung. Um näher an Maschinencode heranzukommen müssten die Terme vom Typ *db_lam* noch in den Code einer virtuellen Maschine wie der CAM (categorical abstract machine) übersetzt werden. Dazu existiert beispielsweise schon eine Formalisierung in Coq, beschrieben in [9].

Der Vorteil einer solchen Formalisierung eines Compilers gegenüber bisherigen Formalisierungen wäre die Semantik mit Substitution. Diese würde sich dank Nominal viel einfacher benutzen lassen, um Beweise über Programme zu führen, als eine Semantik mit Umgebungen. Als Mathematiker ist einem nämlich das Einsetzen eines Argumentes viel mehr vertraut, als das Verstauen eines Arguments an einem Ort, den man später erst wiederzufinden hat. Durch den Compiler würden sich die Resultate dieser Beweise auf den generierten Maschinencode übertragen lassen.

end

Literaturverzeichnis

[1] Christian Urban. Nominal techniques in isabelle/hol. *J. Autom. Reasoning*, 40(4):327–356, 2008.

[2] Christian Urban and Michael Norrish. A formal treatment of the barendregt variable convention in rule inductions. In Randy Pollack, editor, *MERLIN*, pages 25–32. ACM, 2005.

[3] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[4] Lawrence C. Paulson. The foundation of a generic theorem prover. *J. Autom. Reasoning*, 5(3):363–397, 1989.

[5] John Hannan and Frank Pfenning. Compiler verification in lf. In *LICS*, pages 407–418. IEEE Computer Society, 1992.

[6] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[7] Henk Barendregt, S. Abramsky, D. M. Gabbay, T. S. E. Maibaum, and H. P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*, pages 117–309. Oxford University Press, 1992.

[8] Robert Harper, Furio Honsell, and Gordon D. Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, 1993.

[9] Samuel Boutin. Proving correctness of the translation from mini-ml to the cam with the coq proof development system. In *with the Coq Proof Development System. Research report RR-2536, INRIA, Rocquencourt*, 1995.