Der λ-Kalkül

Christoph Kreitz

Institut für Informatik, Universität Potsdam, 14482 Potsdam

Zusammenfassung *Dieser Artikel gibt einen Überblick über den λ-Kalkül. Er ist gedacht als Hintergrundmaterial zur Ergänzung der relativ knappen Abhandlung in der Vorlesung "Einführung in die Theoretische Informatik II".*

1 Der λ-Kalkül – ein Überblick

Seit Beginn des 20. Jahrhunderts wurde eine Vielzahl mathematischer Modelle entwickelt, die ein syntaktisches Schließen über den Wert von Ausdrücken ermöglichen. Manche davon, wie die Turingmaschinen oder die Registermaschinen, orientieren sich im wesentlichen an dem Gedanken einer maschinellen Durchführung von Berechnungen. Andere, wie die μ-rekursiven Funktionen, basieren auf einem mathematischen Verständnis davon, was intuitiv berechenbar ist und wie sich berechenbare Funktionen zu neuen zusammensetzen lassen.

Unter all diesen Kalkülen ist der λ-Kalkül der einfachste. Er kennt nur drei Mechanismen: die Definition einer Funktion (*Abstraktion*), die Anwendung einer Funktion auf ein Argument (*Applikation*) und die Auswertung einer Anwendung, bei der die Definition bekannt ist (*Reduktion*). Dabei werden bei der Auswertung ausschließlich Terme manipuliert, ohne daß hierbei deren Bedeutung in Betracht gezogen wird, was einem Rechner ohnehin unmöglich wäre. Die Syntax und der Berechnungsmechanismus sind also extrem einfach. Dennoch läßt sich zeigen, daß man mit diesen Mechanismen alle berechenbaren Funktionen simulieren kann. Wir wollen die Grundgedanken des λ-Kalküls an einem einfachen Beispiel erläutern.

Beispiel 1 Es ist allgemein üblich, Funktionen dadurch zu charakterisieren, daß man ihr Verhalten auf einem beliebigen Argument x beschreibt. Um also zum Beispiel eine Funktion zu definieren, welche ihr Argument zunächst verdoppelt und anschließend die Zahl *drei* hinzuaddiert, schreibt man kurz:

```
f(x) = 2*x+3
```

Diese Notation besagt, daß das Symbol `f` als Funktion zu interpretieren ist, welche jedem Argument `x` den Wert der Berechnung zuordnet, die durch den Ausdruck `2*x+3` beschrieben ist. Dabei kann für `x` jede beliebige Zahl eingesetzt werden. Um einen konkreten Funktionswert wie zum Beispiel `f(4)` auszurechnen, geht man so vor, daß zunächst jedes Vorkommen des Symbols `x` durch `4` ersetzt wird, wodurch sich der Ausdruck `2*4+3` ergibt. Dieser wird anschließend ausgewertet und liefert das Resultat `11`.

Bei diesem Prozeß spielt das Symbol `f` eigentlich keine Rolle. Es dient nur als Abkürzung für eine Funktionsbeschreibung, welche besagt, daß jedem Argument `x` der Wert `2*x+3` zuzuordnen ist. Im Prinzip müßte es also auch möglich sein, ohne dieses

Symbol auszukommen und die Funktion durch einen Term zu beschreiben, der genau die Abbildung `x` $\mapsto$ `2*x+3` wiederspiegelt.

Genau dies ist die Grundidee des λ-Kalküls. Funktionen lassen sich eindeutig durch Terme beschreiben, die angeben, wie sich die Funktion auf einem beliebigen Argument verhält. Der Name des Arguments ist dabei nur ein Platzhalter, der – so die mathematische Sicht – zur *Abstraktion* der Funktionsbeschreibung benötigt wird. Um diese abstrakte Platzhalterrolle zu kennzeichnen, hat man ursprünglich das Symbol '\' benutzt und geschrieben

```
\x. 2*x+3
```

Dies drückt aus, daß eine Abbildung mit formalem Argument `x` definiert wird, welche als Ergebnis den Wert des Ausdrucks rechts vom Punkt liefert. Später wurde – der leichteren Bezeichnung wegen – das Symbol '\' durch den griechischen Buchstaben λ (*lambda*) ersetzt. In heutiger Notation schreibt man

```
λx. 2*x+3
```

Die Abstraktion von Ausdrücken über formale Parameter ist eines der beiden fundamentalen Grundkonzepte des λ-Kalküls. Es wird benutzt, um Funktionen zu *definieren*. Die Beschreibung des Funktionsverhaltens wird zum Namen der Funktion. Natürlich aber will man definierte Funktionen auch auf bestimmte Eingabewerte *anwenden*. Die Notation hierfür ist denkbar einfach: man schreibt einfach das Argument hinter die Funktion und benutzt Klammern, wenn der Wunsch nach Eindeutigkeit dies erforderlich macht. Um also obige Funktion auf den Wert `4` anzuwenden schreibt man einfach:

```
(λx. 2*x+3)(4)
```

Dabei hätte die Klammerung um die `4` auch durchaus entfallen können. Diese Notation – man nennt sie *Applikation* – besagt, daß die Funktion `f = λx. 2*x+3` auf das Argument `4` angewandt wird. Sie sagt aber nichts darüber aus, welcher Wert bei dieser Anwendung herauskommt. Abstraktion und Applikation sind daher nichts anderes als syntaktische *Beschreibungsformen* für Operationen, die auszuführen sind.

Es hat sich gezeigt, daß durch Abstraktion und Applikation alle Terme gebildet werden können, die man zur Charakterisierung von Funktionen und ihrem Verhalten benötigt. Was bisher aber fehlt, ist die Möglichkeit, Funktionsanwendungen auch *auszuwerten*. Auch dieser Mechanismus ist naheliegend: um eine Funktionsanwendung auszuwerten, muß man einfach die Argumente anstelle der Platzhalter einsetzen. Man berechnet den Wert einer Funktionsanwendung also durch *Reduktion* und erhält

```
2*4+3
```

Bei der Reduktion verschwindet also die Abstraktion `λx` und die Anwendung `(4)` und stattdessen wird im inneren Ausdruck der Platzhalter `4` durch das Argument `4` ersetzt. Diese Operation ist nichts anderes als eine simple Ersetzung von Symbolen durch Terme. Damit ist die Auswertung von Termen ein ganz schematischer Vorgang, der sich durch eine einfache Symbolmanipulation beschreiben läßt.

`(λx. 2*x+3)(4)` $\longrightarrow$ `2*4+3` $\overset{*}{\longrightarrow}$ `11`

Anders als Abstraktion und Applikation ist die durch das Symbol $\longrightarrow$ gekennzeichnete Reduktion kein Mechanismus um Terme zu bilden, sondern um Terme in andere Terme umzuwandeln, die im Sinne der vorgesehenen Bedeutung den gleichen Wert haben.

Durch Abstraktion, Applikation und Reduktion ist der λ-Kalkül vollständig charakterisiert. Weitere Operationen sind nicht erforderlich. So können wir uns darauf konzentrieren, präzise Definitionen für diese Konzepte zu liefern und Aussagen über den *Wert* eines λ-Terms zu beweisen.

Im Gegensatz zur mathematischen Beschreibung des Funktionsverhaltens gibt uns der λ-Kalkül eine *intensionale* Sicht auf Funktionen. λ-Terme beschreiben die *innere* Struktur von Funktionen als Operationen, nicht aber ihr äußeres (*extensionales*) Verhalten. Im λ-Kalkül werden Funktionen als eine *Berechnungsvorschrift* angesehen. Diese erklärt, wie der Zusammenhang zwischen dem Argument einer Funktion und ihrem Resultat zu bestimmen ist, nicht aber, welche mengentheoretischen *Objekte* hinter einem Term stehen. Der λ-Kalkül ist also eine Art *Logik der Berechnung* und gleichzeitig *Grundlage aller funktionalen Programmiersprachen*.

Ein wichtiger Unterschied zu Formalismen wie der Prädikatenlogik erster Stufe ist, daß im λ-Kalkül Funktionen selbst wieder Argumente anderer Funktionen sein dürfen. Ausdrücke wie `(λf.λx. f(x)) (λx. 2*x)` werden im λ-Kalkül durchaus sehr häufig benutzt, denn man könnte ohne sie fast nichts beschreiben. Dagegen sind sie in der Prädikatenlogik erster Stufe verboten. In diesem Sinne ist der λ-Kalkül eine *Logik höherer Ordnung*.

Aus der Berechnungsvorschrift des λ-Kalküls ergibt sich unmmittelbar auch ein logischer Kalkül zum Schließen über den Wert eines λ-Ausdrucks. Logisch betrachtet ist damit der λ-Kalkül ein *Kalkül der Gleichheit* und er bietet ein klar definiertes und einfaches Konzept, um logische Schlußfolgerungen über das Verhalten von Programmen zu ziehen. Es muß nur sichergestellt werden, daß sich die Gleichheit zweier Terme genau dann beweisen läßt, wenn die Berechnungsvorschrift bei beiden zum gleichen Ergebnis führt.

Die Semantik von λ-Ausdrücken mengentheoretisch zu beschreiben ist dagegen verhältnismäßig schwierig, da – anders als bei der Prädikatenlogik – eine konkrete mengentheoretische Semantik bei der Entwicklung des λ-Kalküls keine Rolle spielte. Zwar ist es klar, daß es sich bei λ-Ausdrücken im wesentlichen um Funktionen handeln soll, aber die zentrale Absicht war die Beschreibung der *berechenbaren* Funktionen. Eine *operationale* Semantik, also eine Vorschrift, wie man den Wert eines Ausdrucks ausrechnet, läßt sich daher leicht angeben. Was aber die berechenbaren Funktionen im Sinne der Mengentheorie genau sind, das kann man ohne eine komplexe mathematische Theorie kaum angeben. Es ist daher kein Wunder, daß die (extensionale) Semantik erst lange nach der Entwicklung des Kalküls gegeben werden konnte.

Wir werden im folgenden zuerst die Syntax von λ-Ausdrücken sowie ihre operationale Semantik beschreiben und die mengentheoretische Semantik nur am Schluß kurz skizzieren. Wir werden zeigen, daß der λ-Kalkül tatsächlich genauso ausdrucksstark ist wie jedes andere Berechenbarkeitsmodell und hierfür eine Reihe von Standardoperationen durch λ-Ausdrücke beschreiben. Die Turing-Mächtigkeit des λ-Kalküls hat natürlich auch Auswirkungen auf die Möglichkeiten einer automatischen Unterstützung des Kalküls zum Schließen über die Werte von λ-Ausdrücken. Dies werden wir am Ende dieses Artikels besprechen.

2 Syntax

λ-Ausdrücke bilden im Prinzip eine Programmiersprache mit einer extrem einfachen Syntax , da es nur Variablen, Abstraktion und Applikation gibt.

Definition 2 (λ-Terme).
Es sei $\mathcal{V}$ ein Alphabet von Variablensymbolen. Die Terme der Sprache des λ-Kalküls, kurz λ-Terme, sind induktiv wie folgt definiert.

- *Jede Variable $x \in \mathcal{V}$ ist ein λ-Term.*
- *Die λ-Abstraktion $\lambda x\,.\,t$ ein λ-Term, wenn $x \in \mathcal{V}$ eine Variable und t ein λ-Term ist.*
- *Die Applikation $f\ t$ ein λ-Term, wenn t und f λ-Terme sind.*
- *(t) ist ein λ-Term, wenn t ein λ-Term ist.*

Für die Wahl der Variablennamen gibt es im Prinzip keinerlei Einschränkungen bis auf die Tatsache, daß sie sich im Computer darstellen lassen sollten und keine reservierten Symbole wie λ enthalten dürfen. Namenskonventionen zur Unterscheidung von Objekten und Funktionen lassen sich nicht gut einhalten, da bei einem Term wie `λx.x` noch nicht feststeht, ob die Variable `x` ein einfaches Objekt oder eine Funktion beschreiben soll. Für die Verarbeitung ist dies ohnehin unerheblich. Wir wollen die Bildung von Termen gemäß der obigen Regeln nun anhand einiger Beispiele veranschaulichen.

Beispiel 3 *Die folgenden Ausdrücke sind korrekte λ-Terme im Sinne von Definition 2.*

1. `x` *und* `pair` — *Alle "Symbole" sind Variablen*
2. `λf.λx.f(x)` — *Anwendung einer Funktion*
3. `λf.λg.λx. f g (g x)` — *Funktionen höherer Ordnung*
4. `x(x)` *und* `(λx.x(x))(λx.x(x))` — *Selbstanwendung*

Die Beispiele zeigen, daß es auch erlaubt ist, Terme zu bilden, bei denen man Funktionen *höherer Ordnung* – also Funktionen, deren Argumente wiederum Funktionen sind, wie z.B. `f` in `f g (g x)` – assoziiert, und Terme, die *Selbstanwendung* wie in `x(x)` beschreiben. Dies macht die Beschreibung einer mengentheoretischen Semantik (siehe Abschnitt 6) natürlich viel schwieriger als zum Beispiel bei der Prädikatenlogik.

Die Definition läßt es zu, λ-Terme zu bilden, ohne daß hierzu Klammern verwendet werden müssen. Dadurch stellt sich jedoch die Frage nach einer eindeutigen Decodierbarkeit ungeklammerter λ-Terme. Deshalb müssen wir Prioritäten einführen und vereinbaren, unter welchen Voraussetzungen Klammern entfallen fürfen.

Definition 4 (Konventionen zur Eindeutigkeit von λ-Termen).
Die Applikation bindet stärker als die λ-Abstraktion und ist linksassoziativ. In einem λ-Term muß ein durch einen stärker bindenden Operator gebildeter Teilterm nicht geklammert werden.

Gemäß dieser Konvention ist also ein Term der Form $f\,a\,b$ gleichbedeutend mit $(f(a))(b)$ und nicht etwa mit $f(a(b))$. Die Konvention, die Applikation linksassoziativ zu interpretieren, liegt darin begründet, daß hierdurch der Term $f\,a\,b$ etwa dasselbe Verhalten zeigt wie die Anwendung einer Funktion f auf das Paar (a, b). Bei letzterer werden die Argumente auf einmal verarbeitet, während sie im λ-Kalkül der Reihe

nach abgearbeitet werden: Die Anwendung von f auf a liefert eine neue Funktion, die wiederum auf b angewandt wird. Obwohl die λ-Abstraktion nur für *einstellige* Funktionen definiert ist, können mehrstellige Funktionen also problemlos simuliert werden. Entsprechend verwendet man im λ-Kalkül die Notation $f(t_1, \ldots, t_n)$ oft als Abkürzung fuer die Applikation $f\ t_1 \ldots t_n$. [1]

Eine Assoziativitätsvereinbarung für die Abstraktion braucht nicht getroffen zu werden. Der Term $\lambda x.\lambda y.t$ ist gleichwertig mit $\lambda x.(\lambda y.t)$, da die alternative Klammersetzung $(\lambda x.\lambda y).t$ keinen syntaktisch korrekten λ-Term ergibt.

3 Operationale Semantik: Auswertung von Termen

In Beispiel 1 haben wir bereits angedeutet, daß wir mit jedem λ-Term natürlich eine gewisse Bedeutung assoziieren. Der Term `λx.2*x` soll eine Funktion beschreiben, die bei Eingabe eines beliebigen Argumentes dieses verdoppelt. Diese Bedeutung wird im λ-Kalkül durch eine *Berechnungsvorschrift* ausgedrückt, die aussagt, auf welche Art der Wert eines λ-Terms zu bestimmen ist. Diese Vorschrift verwandelt den bisher bedeutungslosen λ-Kalkül in einen Berechnungsformalismus.

Es ist offensichtlich, daß eine Berechnungsregel rein syntaktischer Natur sein muß, denn Rechenmaschinen können ja nur Symbole in andere Symbole umwandeln. Im Falle des λ-Kalküls ist diese Regel sehr einfach: Wird die Applikation einer Funktion $\lambda x.t$ auf ein Argument s ausgewertet, so wird der formale Parameter x der Funktion – also die Variable der λ-Abstraktion – im Funktionskörper t durch das Argument s ersetzt. Der Ausdruck $(\lambda x.t)(s)$ wird also zu $t[s/x]$ *reduziert*. Um dies präzise genug zu definieren müssen wir Definitionen für freie und gebundene Vorkommen von Variablen und die Substitution von Variablen durch Terme angeben.

Definition 5 (Freies und gebundenes Vorkommen von Variablen).
Es seien $x, y \in \mathcal{V}$ Variablen sowie f und t λ-Terme. Das freie und gebundene Vorkommen der Variablen x in einem λ-Term ist induktiv durch die folgenden Bedingungen definiert.

1. *Im λ-Term x kommt x frei vor. Die Variable y kommt nicht vor, wenn x und y verschieden sind.*
2. *In $\lambda x.t$ wird jedes freie Vorkommen von x in t gebunden. Gebundene Vorkommen von x in t bleiben gebunden. Jedes freie Vorkommen der Variablen y bleibt frei, wenn x und y verschieden sind.*
3. *In $f\ t$ bleibt jedes freie Vorkommen von x in f oder t frei und jedes gebundene Vorkommen von x in f oder t gebunden.*
4. *In (t) bleibt jedes freie Vorkommen von x in t frei und jedes gebundene Vorkommen gebunden.*

Das freie Vorkommen der Variablen $x_1, .., x_n$ in einem λ-Term t wird mit $t[x_1, .., x_n]$ gekennzeichnet. Ein λ-Term ohne freie Variablen heißt geschlossen.

[1] Die Umwandlung einer Funktion f, die auf Paaren von Eingaben operiert, in eine einstellige Funktion höherer Ordnung F mit der Eigenschaft $F(x)(y) = f(x, y)$ nennt man – in Anlehnung an den Mathematiker Haskell B. Curry – *currying*. Es sei allerdings angemerkt, daß diese Technik auf den Mathematiker Schönfinkel und nicht etwa auf Curry zurückgeht.

Man beachte, daß Variablen auch innerhalb des "Funktionsnamens" einer Applikation $f\ t$ frei oder gebunden vorkommen können, da f seinerseits ein komplexerer λ-Term sein kann. Das folgende Beispiel illustriert die Möglichkeiten des freien bzw. gebundenen Vorkommens von Variablen.

Beispiel 6 *Wir analysieren das Vorkommen von* `x` *im* λ*-Term* `λf.λx.(λz.f x z) x`.

- *Die Variable* `x` *tritt frei im* λ*-Term* `x` *auf.*
- *Nach (3.) ändert sich nichts, wenn die* λ*-Terme* `f x` *und* `f x z` *gebildet werden.*
- *Nach (2.) bleibt* `x` *frei im* λ*-Term* `λz. f x z`.
- *Nach (3.) bleibt* `x` *frei im* λ*-Term* `(λz. f x z) x`.
- *In* `λx.(λz. f x z) x` *ist* `x` *gemäß (2.) – von außen betrachtet – gebunden.*
- *Nach (2.) bleibt* `x` *gebunden im gesamten Term* `λf.λx.(λz. f x z) x`.

Insgesamt haben wir folgende Vorkommen von `x`*:* $\lambda f.\lambda x.\overbrace{(\lambda z.\ f\ x\ z)}^{x\ \textit{frei}}\ \overbrace{x}^{x\ \textit{gebunden}}$

Das Konzept der Substitution ist eine Erweiterung der bekannten Substitution aus der Prädikatenlogik. Während in der Prädikatenlogik Ersetzungen nur innerhalb der Argumente einer Funktionsanwendung durchgeführt werden dürfen, können im λ-Kalkül auch innerhalb der Funktion selbst, die ebenfalls ein Term ist, Ersetzungen vorgenommen werden. Die λ-Abstraktion dagegen verhält sich wie ein Quantor. Entsprechend müssen wir drei Fälle betrachten.

Definition 7 (Substitution).
Eine Substitution ist eine endliche Abbildung σ *von der Menge* $\mathcal{V}$ *der Variablen in die Menge der Terme (d.h.* $\sigma(x)\neq x$ *gilt nur für endlich viele Variablen* $x\in\mathcal{V}$*). Für* $\sigma(x_1)=t_1,\ldots,\sigma(x_n)=t_n$ *schreiben wir kurz* $\sigma = [t_1,\ldots,t_n/x_1,\ldots,x_n]$.
$u[t/x]$ *bezeichnet die Anwendung einer Substitution* $\sigma=[t/x]$ *auf einen* λ*-Term* u.[2] *Diese ist induktiv wie folgt definiert.*

$$
\begin{array}{lll}
x[t/x] & = & t \\
x[t/y] & = & x, \quad \textit{wenn } x \textit{ und } y \textit{ verschieden sind.} \\
(\lambda x\,.\,u)[t/x] & = & \lambda x.u \\
(\lambda x\,.\,u)[t/y] & = & (\lambda z\,.\,u[z/x])[t/y], \quad \textit{wenn } x \textit{ und } y \textit{ verschieden sind, } y \textit{ frei in } u \textit{ vorkommt und } x \textit{ frei in } t \textit{ vorkommt. } z \textit{ ist eine neue Variable, die von } x \textit{ und } y \textit{ verschieden ist und weder in } u \textit{ noch in } t \textit{ vorkommt.} \\
(\lambda x\,.\,u)[t/y] & = & \lambda x.u[t/y], \quad \textit{wenn } x \textit{ und } y \textit{ verschieden sind und es der Fall ist, daß } y \textit{ nicht frei in } u \textit{ ist oder daß } x \textit{ nicht frei in } t \textit{ vorkommt.} \\
(f\ u)[t/x] & = & f[t/x]\ u[t/x] \\
(u)\,[t/x] & = & (u[t/x])
\end{array}
$$

Dabei sind $x, y\in\mathcal{V}$ *Variablen sowie* f,u *und* t λ*-Terme.*

[2] Leider gibt es eine Fülle von Notationen für die Substitution. Anstelle von $\sigma(x)=t$ schreiben manche $\sigma=[x\backslash t]$ andere $\sigma=\{x\backslash t\}$ oder $\sigma=\{t/x\}$. Die hier verwendete Konvention $\sigma=[t/x]$ ist in der Literatur zum λ-Kalkül und zur Programmierung üblich. All diesen Schreibweisen ist aber gemein, daß die zu ersetzende Variable "unterhalb" des Terms steht, der sie ersetzt. Auch hat es sich eingebürgert, die Anwendung einer Substitution auf einen Ausdruck dadurch zu kennzeichnen, daß man die Substitution *hinter* den Ausdruck schreibt.

Die Anwendung komplexer Substitutionen der Form $u[t_1, .., t_n/x_1, .., x_n]$ wird entsprechend definiert. Wir wollen Substitution an einem einfachen Beispiel erklären.

Beispiel 8

$$
\begin{aligned}
&\lfloor \lambda f.\lambda x.\ n\ f\ (f\ x) \rfloor [\lambda f.\lambda x.x/n] \\
=\ &\lambda f.\ \lfloor \lambda x.\ n\ f\ (f\ x) \rfloor [\lambda f.\lambda x.x/n] \\
=\ &\lambda f.\lambda x.\ \lfloor n\ f\ (f\ x) \rfloor [\lambda f.\lambda x.x/n] \\
=\ &\lambda f.\lambda x.\ \lfloor n\ f \rfloor [\lambda f.\lambda x.x/n]\ \lfloor (f\ x) \rfloor [\lambda f.\lambda x.x/n] \\
=\ &\lambda f.\lambda x.\ n[\lambda f.\lambda x.x/n]\ f[\lambda f.\lambda x.x/n]\ (\lfloor f\ x \rfloor [\lambda f.\lambda x.x/n]) \\
=\ &\lambda f.\lambda x.\ (\lambda f.\lambda x.x)\ f\ (\lfloor f \rfloor [\lambda f.\lambda x.x/n]\ \lfloor x \rfloor [\lambda f.\lambda x.x/n]) \\
=\ &\lambda f.\lambda x.\ (\lambda f.\lambda x.x)\ f\ (f\ x) \\
=\ &\lambda f.\lambda x.\ (\lambda f.\lambda x.x)\ f\ (f\ x) \\
=\ &\lambda f.\lambda x.\ (\lambda f.\lambda x.x)\ f\ (f\ x)
\end{aligned}
$$

Bei der Anwendung einer Substitution auf eine λ-Abstraktion müssen wir also in besonderen Fällen wiederum auf eine Umbenennung gebundener Variablen zurückgreifen, um zu vermeiden, daß eine freie Variable ungewollt in den Bindungsbereich der λ-Abstraktion gerät. Diese Umbenennung gebundener Variablen ändert die Bedeutung eines Terms überhaupt nicht. Terme, die sich nur in den Namen ihrer gebundenen Variablen unterscheiden, müssen also im Sinne der Semantik als gleich angesehen werden, auch wenn sie textlich verschieden sind. Zur besseren Unterscheidung nennt man solche Termpaare daher *kongruent*.

Definition 9 (α-Konversion).
Eine Umbenennung gebundener Variablen, oder α-Konversion, in einem λ-term t ist die Ersetzung eines Teilterms der Gestalt $\lambda x.u$ durch den Term $\lambda z.u[z/x]$, wobei z eine Variable ist, die in u bisher nicht vorkam.
Zwei λ-Terme t und u heißen kongruent oder α-konvertibel, wenn u sich aus t durch endlich viele Umbenennungen gebundener Variablen ergibt.

Umbennenung kann also als eine erste einfache Rechenvorschrift angesehen werden, welche Terme in andere Terme *konvertiert* (umwandelt), die syntaktisch verschieden aber gleichwertig sind. Eine wirkliche Auswertung eines Termes findet jedoch nur statt, wenn reduzierbare Teilterme (Englisch: *red*ucible *ex*pression, oder kurz *Redex*) durch ihre kontrahierte Form, das sogenannte *Kontraktum* ersetzt werden.

Definition 10 (Reduktion).
Die Reduktion eines λ-terms t ist die Ersetzung eines Teiltermes der Gestalt $(\lambda x.u)(s)$ durch den Term $u[s/x]$. $(\lambda x.u)(s)$ wird dabei als Redex bezeichnet und $u[s/x]$ als sein Kontraktum.
Ein λ-Term t heißt reduzierbar auf einen λ-Term u, im Zeichen $t \xrightarrow{*} u$, wenn u sich aus t durch endlich viele Reduktionen und α-Konversionen ergibt.

Für die Auswertung von λ-Termen durch Menschen würden diese Definitionen vollkommen ausreichen. Da wir den Prozeß der Reduktion jedoch als Berechnungsmechanismus verstehen wollen, den eine Maschine ausführen soll, geben wir zusätzlich eine detaillierte formale Definition.

Definition 11 (Formale Definition der Reduzierbarkeit).

Konversion von λ-Termen ist eine binäre Relation $\cong$, die induktiv wie folgt definiert ist.

1. $\lambda x\,.\,u \cong \lambda z\,.\,u[z/x]$, *falls die Variable* z *nicht in* u *vorkommt.* — α-Konversion
2. $f\,t \cong f\,u$, *falls* $t \cong u$ — μ-Konversion
3. $f\,t \cong g\,t$, *falls* $f \cong g$ — ν-Konversion
4. $\lambda x\,.\,t \cong \lambda x\,.\,u$, *falls* $t \cong u$ — ξ-Konversion
5. $t \cong t$ — ρ-Konversion
6. $t \cong u$, *falls* $u \cong t$ — σ-Konversion
7. $t \cong u$, *falls es einen λ-Term s gibt mit* $t \cong s$ *und* $s \cong u$ — τ-Konversion

Reduktion ist eine binäre Relation $\longrightarrow$, die induktiv wie folgt definiert ist.

1. $(\lambda x\,.\,u)\;s \longrightarrow u[s/x]$ — β-Reduktion
2. $f\,t \longrightarrow f\,u$, *falls* $t \longrightarrow u$ — μ-Reduktion
3. $f\,t \longrightarrow g\,t$, *falls* $f \longrightarrow g$ — ν-Reduktion
4. $\lambda x\,.\,t \longrightarrow \lambda x\,.\,u$, *falls* $t \longrightarrow u$ — ξ-Reduktion
5. $t \longrightarrow u$, *falls es ein s gibt mit* $t \longrightarrow s$ *und* $s \cong u$, *oder* $t \cong s$ *und* $s \longrightarrow u$

Die Relation $\xrightarrow{n}$ ist induktiv wie folgt definiert.

1. $t \xrightarrow{0} u$, *falls* $t \cong u$
2. $t \xrightarrow{n+1} u$, *falls es einen λ-Term s gibt mit* $t \longrightarrow s$ *und* $s \xrightarrow{n} u$

Reduzierbarkeit von λ-Termen ist eine binäre Relation $\xrightarrow{*}$, die definiert ist durch $t \xrightarrow{*} u$, *falls* $t \xrightarrow{n} u$ *für ein* $n \in \mathbb{N}$.

Die Regeln, die zusätzlich zur zur β-Reduktion bzw. zur α-Konversion genannt werden, drücken aus, daß Reduktion auf jedes Redex innerhalb eines Termes angewandt werden darf, um diesen zu verändern. Unter all diesen Regeln ist die β-Reduktion die einzige 'echte' Reduktion. Aus diesem Grunde schreibt man oft auch $t \xrightarrow{\beta} u$ anstelle von $t \longrightarrow u$. Wir wollen nun die Reduktion an einigen Beispielen erklären.

Beispiel 12

1. *Wir reduzieren den Term* `(λn.λf.λx. n f (f x)) (λf.λx.x)`.
   Der erste Schritt ist die β-Reduktion
   `(λn.λf.λx.n f (f x))(λf.λx.x)` $\xrightarrow{\beta}$ `⌊λf.λx.n f (f x)⌋[(λf.λx.x)/n]`,
   was nach Beispiel 8 auf Seite 7 den Term `λf.λx. (λf.λx.x) f (f x)` *liefert. Wir müssen nun die inneren Terme reduzieren.*

es gilt `(λf.λx.x) f`	$\xrightarrow{\beta}$ `⌊λx.x⌋[f/f]` = `λx.x`		β
also `(λf.λx.x) f (f x)`	$\longrightarrow$ `(λx.x) (f x)`		μ
also `λx.(λf.λx.x) f (f x)`	$\longrightarrow$ `λx.(λx.x) (f x)`		ξ
also `λf.λx.(λf.λx.x) f (f x)`	$\longrightarrow$ `λf.λx.(λx.x) (f x)`		ξ

es gilt `(λx.x) (f x)`	$\xrightarrow{\beta}$ `⌊x⌋[f x/x]` = `f x`		β
also `λx.(λx.x) (f x)`	$\longrightarrow$ `λx. f x`		ξ
also `λf.λx.(λx.x) (f x)`	$\longrightarrow$ `λf.λx. f x`		ξ

   Der Term `λf.λx. f x` *läßt sich nicht weiter reduzieren.*

Diese ausführliche Beschreibung zeigt alle Details einer formalen Reduktion einschließlich des Hinabsteigens in Teilterme, in denen sich die zu kontrahierenden Redizes befinden. Diese Form ist für eine Reduktion "von Hand" jedoch zu ausführlich, da es einem Menschen nicht schwerfällt, einen zu reduzierenden Teilterm zu identifizieren und die Reduktion durchzuführen. Wir schreiben daher kurz

```
    (λn.λf.λx. n f (f x)) (λf.λx.x)
⟶ λf.λx. (λf.λx.x) f (f x)
⟶ λf.λx. (λx.x) (f x)
⟶ λf.λx. f x
```

oder noch kürzer: $(\lambda n.\lambda f.\lambda x.\, n\ f\ (f\ x))\ (\lambda f.\lambda x.x) \xrightarrow{3} \lambda f.\lambda x.\, f\ x$

2. *Der Term* $(\lambda n.\lambda f.\lambda x.\, n\ f\ (f\ x))\ (\lambda f.\lambda x.\, f\ x)$ *wird ähnlich reduziert*

```
    (λn.λf.λx. n f (f x)) (λf.λx. f x)
⟶ λf.λx. (λf.λx. f x) f (f x)
⟶ λf.λx. (λx. f x) (f x)
⟶ λf.λx. f (f x)
```

3. *Bei der Reduktion des Terms* $(\lambda f.\lambda x.\, f\ x\ (f\ f))\ (\lambda x.\lambda y.\, y)$ *gibt es mehrere Möglichkeiten vorzugehen. Die vielleicht naheliegendste ist die folgende:*

```
    (λf.λx. f x (f f)) (λx.λy. y)
⟶ λx. (λx.λy. y) x ((λx.λy. y) (λx.λy. y))
⟶ λx. (λy. y) ((λx.λy. y) (λx.λy. y))
⟶ λx. ((λx.λy. y) (λx.λy. y)
⟶ λx. λy. y
```

Wir hätten ab dem zweiten Schritt aber auch zunächst das rechte Redex reduzieren können und folgende Reduktionskette erhalten:

```
    (λf.λx. f x (f f)) (λx.λy. y)
⟶ λx. (λx.λy. y) x ((λx.λy. y) (λx.λy. y))
⟶ λx. (λx.λy. y) x (λy. y)
⟶ λx. (λy. y) (λy. y)
⟶ λx. λy. y
```

Das dritte Beispiel zeigt, daß Reduktion keineswegs ein deterministischer Vorgang ist. Unter Umständen gibt es mehrere Redizes, auf die eine β-Reduktion angewandt werden kann, und damit auch mehrere Arten, den Wert eines Terms durch Reduktion auszurechnen. Um also den λ-Kalkül als eine Art Programmiersprache verwendbar zu machen, ist es nötig, eine *Reduktionsstrategie* zu fixieren. Diese Strategie muß beim Vorkommen mehrerer Redizes in einem Term entscheiden, welches von diesen zuerst durch ein Kontraktum ersetzt wird. Die Auswirkungen einer solchen Festlegung und weitere Reduktionseigenschaften des λ-Kalküls werden wir im Abschnitt 7 diskutieren.

Die Reduktion von λ-Termen dient dazu, den Wert eines gegebenen Termes zu bestimmen. Reduktion alleine reicht aber nicht immer aus, wenn man das Verhalten von λ-Programmen analysieren will, da man nicht nur Terme untersuchen möchte, die – wie 4+5 und 9 – aufeinander reduzierbar sind, sondern auch Terme, die – wie 4+5 und 2+7 – den gleichen Wert haben. Mit den bisher eingeführten Konzepten läßt sich Werte-Gleichheit relativ leicht definieren.

Definition 13 (Gleichheit von λ-Termen).
Zwei λ-Terme heißen semantisch gleich (konvertierbar), wenn sie auf denselben λ-Term reduziert werden können:

$$t{=}u \text{ gilt genau dann, wenn es einen Term } v \text{ gibt mit } t \xrightarrow{*} v \text{ und } u \xrightarrow{*} v.$$

Man beachte, daß Werte-Gleichheit mehr als nur syntaktische Gleichheit ist und daß sich das Gleichheitssymbol = im folgenden immer auf diese Werte-Gleichheit bezieht.

4 Vom λ-Kalkül zu echten Programmen

Bisher haben wir den λ-Kalkül im wesentlichen als ein formales System zur Manipulation von Termen betrachtet. Wir wollen nun zeigen, daß dieser einfache Formalismus tatsächlich geeignet ist, bekannte berechenbare Funktionen auszudrücken. Dabei müssen wir jedoch beachten, daß die einzige Berechnungsform die Reduktion von Funktionsanwendungen ist. Zahlen, boolesche Werte, Datencontainer und ähnliche aus den meisten Programmiersprachen vertraute Operatoren gehören nicht zu den Grundkonstrukten des λ-Kalküls. Wir müssen sie daher im λ-Kalkül *simulieren*.[3]

Aus theoretischer Sicht bedeutet dies, den λ-Kalkül durch Einführung abkürzender Definitionen *konservativ* zu erweitern. Auf diese Art behalten wir die einfache Grundstruktur des λ-Kalküls, wenn es darum geht, grundsätzliche Eigenschaften dieses Kalküls zu untersuchen, erweitern aber gleichzeitig die Ausdruckskraft der vordefinierten formalen Sprache und gewinnen somit an Flexibilität.

In diesem Sinne sind höhere funktionale Programmiersprachen wie LISP oder ML nichts anderes als konservative Erweiterungen des λ-Kalküls, während dieser als eine Art *Assemblersprache funktionaler Programmierung* angesehen werden kann. [4] Der wesentliche Unterschied liegt nur in der Menge der vordefinierten Konstrukte, die ein einfacheres Programmieren in diesen Sprachen erst möglich machen.

Wir werden im folgenden zeigen, wie die wichtigsten Daten- und Programmstrukturen im λ-Kalkül simuliert werden können. Dabei geht weniger darum eine möglichst effiziente Repräsentation zu finden, als darum, zeigen zu können, daß Berechnungen auf den Repräsentation der entsprechenden Konstrukte genau die Eigenschaften dieser Konstrukte widerspiegeln. So muß z.B. bei booleschen Operationen eine Fallunterscheidung if b then s else t angewandt auf den booleschen Wert T, den Term s und angewandt auf den booleschen Wert F, den Term t zurückgeben, eine Projektionsfunktion angewandt auf ein Paar $(s\,,t)$ die erste bzw. zweite Komponente dieses Paars liefern, eine Simulation der Addition auf der Repräsentation zweier Zahlen die Repräsentation ihrer Summer berechnen, u.s.w.

[3] In *realen* Computersystemen ist dies nicht anders. Auch ein Computer operiert nicht auf Zahlen, sondern nur auf Bitmustern, welche Zahlen *darstellen*. Alle arithmetischen Operationen, die ein Computer ausführt, sind nichts anderes als eine Simulation dieser Operationen durch entsprechende Manipulationen der Bitmuster, während die meisten Programmstrukturen im wesentlichen durch Sprungbefehle simuliert werden.

[4] Es gibt Spezialhardware (Lisp-Maschinen), mit denen man λ-Terme auch direkt, also ohne Umweg über eine Simulation in der konventionellen Von-Neumann Architektur, auswerten kann. Diese hat sich jedoch aus wirtschaftlichen Gründen nicht durchgesetzt.

4.1 Boolesche Operatoren

Im reinen λ-Kalkül ist die Funktionsanwendung (Applikation) die einzige Möglichkeit, "Programme" und "Daten" in Verbindung zu bringen. In praktischen Anwendungen besteht jedoch oft die Notwendigkeit, bestimmte Programmteile nur auszuführen, wenn eine Bedingung erfüllt ist. Um dies zu simulieren, benötigt man zwei boolesche Werte T und F sowie ein *Conditional* $\mathsf{cond}(b;s;t)$ zur Erzeugung *bedingter Funktionsaufrufe*. Die folgende Repräsentation stellt sicher, daß die beiden Werte verschieden sind und daß das Conditional diese beiden Werte unterscheidet.

Definition 14 (Repräsentation boolescher Operatoren).

$$\begin{array}{ll} \mathsf{T} & \equiv \lambda x.\lambda y.x \\ \mathsf{F} & \equiv \lambda x.\lambda y.y \\ \mathsf{cond}(b;s;t) & \equiv b\,s\,t \end{array}$$

In der obigen Definition wird der Ausdruck auf der linken Seite des $\equiv$-Symbols durch den λ-Term der rechten Seite repräsentiert. Enthält die linke Seite *Metavariablen*, also Platzhalter für andere Terme (kursiv und rot markiert), so werden konkrete Terme an dieser Stelle einfach an die enstprechende Stelle auf der rechten Seite übertragen.

Die Terme T und F sollen die Wahrheitswerte *wahr* und *falsch* simulieren. Der Term $\mathsf{cond}(b;s;t)$ nimmt einen booleschen Ausdruck b als erstes Argument, wertet diesen aus und berechnet dann – je nachdem ob diese Auswertung T oder F ergab – den Wert von s oder den von t. Es ist nicht schwer zu beweisen, daß T, F und cond sich wie die entsprechenden booleschen Operatoren verhalten, die man hinter dem Namen vermutet.

Beispiel 15 *Wir zeigen, daß* $\mathsf{cond}(\mathsf{T};s;t)$ *zu* s *reduziert und* $\mathsf{cond}(\mathsf{F};s;t)$ *zu* t.

$$\begin{array}{l} \mathsf{cond}(\mathsf{T};s;t) \equiv \mathsf{T}\,s\,t \equiv (\lambda x.\lambda y.x)\,s\,t \longrightarrow (\lambda y.s)\,t \longrightarrow s \\ \mathsf{cond}(\mathsf{F};s;t) \equiv \mathsf{F}\,s\,t \equiv (\lambda x.\lambda y.y)\,s\,t \longrightarrow (\lambda y.y)\,t \longrightarrow t \end{array}$$

Damit ist die Repräsentation des Conditionals tatsächlich invers zu der von T und F. Da die λ-Terme fü T und F auch fundamental unterschiedliche Funktionen beschreiben, kann man mit ihnen den Gedanken ausdrücken, daß T und F einander widersprechen.

Die bisherige Präsentationsform des Conditionals als Operator cond, der drei Eingabeparameter erwartet, entspricht immer der Denkweise von Funktionsdefinition und -anwendung. Sie hält sich daher an die syntaktischen Einschränkungen an den Aufbau von Termen, die in der Prädikatenlogik üblich und für Parser leicht zu verarbeiten ist. Für viele Programmierer ist diese Präsentationsform jedoch schwer zu lesen, da sie mit der Schreibweise $\mathsf{if}\ b\ \mathsf{then}\ s\ \mathsf{else}\ t$ vertrauter sind. Wir ergänzen daher die Definitionen boolescher Operatoren um eine verständlichere Notation für das Conditional.

$$\mathsf{if}\ b\ \mathsf{then}\ s\ \mathsf{else}\ t \equiv \mathsf{cond}(b;s;t)$$

Im folgenden werden wir für die meisten Operatoren zwei Formen definieren: eine abstrakte 'Termform', die für eine Verarbeitung im Computer besser geeignet ist, sowie eine für Menschen leichter zu lesende 'Display Form' dieses Terms.

4.2 Paare: Datenkapselung und Komponentenzugriff

Boolesche Operationen wie das Conditional können dazu benutzt werden, 'Programme' besser zu strukturieren. Aber auch bei den 'Daten' ist es wünschenswert, Strukturierungsmöglichkeiten anzubieten. Die wichtigste dieser Strukturierungsmöglichkeiten ist die Bildung von Tupeln, die es uns erlauben, $f(a,b,c)$ anstelle von $f\,a\,b\,c$ zu schreiben. Auf diese Art wird deutlicher, daß die Werte a, b und c zusammengehören und nicht etwa einzeln abgearbeitet werden sollen.

Die einfachste Form von Tupeln sind Paare, die wir durch den abstrakten Term $\mathsf{pair}(a,b)$ repräsentieren und in der bekannten Form (a,b) darstellen. Komplexere Tupel können durch das Zusammensetzen von Paaren gebildet werden. (a,b,c) läßt sich zum Beispiel als $(a,(b,c))$ schreiben. Neben der *Bildung* von Paaren aus einzelnen Komponenten benötigen wir natürlich auch Operatoren, die ein Paar p *analysieren*. Üblicherweise verwendet man hierzu *Projektionen*, die wir mit $p.1$ und $p.2$ bezeichnen und durch die abstrakten Terme $\mathsf{pi1}(p)$ und $\mathsf{pi2}(p)$ darstellen.

Definition 16 (Operatoren auf Paaren).

$$
\begin{array}{lll}
(u,v) & \equiv \mathsf{pair}(u,v) & \equiv \lambda\mathtt{pair.\ pair}\ u\ v \\
pair.1 & \equiv \mathsf{pi1}(pair) & \equiv pair\ \ (\lambda\mathtt{x.}\lambda\mathtt{y.x}) \\
pair.2 & \equiv \mathsf{pi2}(pair) & \equiv pair\ \ (\lambda\mathtt{x.}\lambda\mathtt{y.y}) \\
\mathsf{let}\ \ (x,y)=pair\ \mathsf{in}\ \ t & \equiv \mathsf{spread}(pair\,;x,y.t) & \equiv pair\ \ (\lambda x.\lambda y.t)
\end{array}
$$

Der spread-Operator beschreibt eine *einheitliche* Möglichkeit, Paare zu analysieren: ein Paar p wird aufgespalten in zwei Komponenten, die wir mit x und y bezeichnen und in einem Term t weiter verarbeiten. [5] Die Projektionen können als Spezialfall des spread-Operators betrachtet werden, in denen der Term t entweder x oder y ist.

In seiner ausführlicheren Notation $\mathsf{let}\ \ (x,y)=p\ \mathsf{in}\ \ t$ wird dieses Konstrukt zum Beispiel in der Sprache ML benutzt. Bei seiner Ausführung wird der Term p ausgewertet bis seine beiden Komponenten feststehen. Diese werden dann anstelle der Variablen x und y im Term t eingesetzt. Das folgende Beispiel zeigt, daß genau dieser Effekt durch die obige Definition erreicht wird.

Beispiel 17 *Wir zeigen, daß der* spread*-Operator invers zur Paarbildung ist.*

$$
\begin{array}{lll}
\mathsf{let}\ \ (x,y)=(u,v)\ \mathsf{in}\ \ t & \equiv & (u,v)(\lambda x.\lambda y.t) \\
& \equiv & (\lambda\mathtt{pair.\ pair}\ u\ v)(\lambda x.\lambda y.t) \\
& \longrightarrow & (\lambda x.\lambda y.t)\ u\ v \\
& \longrightarrow & (\lambda y.t[u/x])\ v \\
& \longrightarrow & t[u,v/x,y]
\end{array}
$$

Auf ähnliche Weise kann man zeigen $(u,v).1 \xrightarrow{*} u$ und $(u,v).2 \xrightarrow{*} v$.

[5] Aus logischer Sicht sind x und y zwei Variablen, deren Vorkommen in t durch den spread-Operator gebunden wird. Zusätzlich wird festgelegt, daß x an die erste Komponente des Paares p gebunden wird und y an die zweite.

4.3 Natürliche Zahlen

Es gibt viele Möglichkeiten, natürliche Zahlen und Operationen auf Zahlen im λ-Kalkül darzustellen. Die einfachsten Repräsentationen beschränken sich darauf, Zahlen durch iterierte Terme zu beschreiben, so daß man die Größe der Zahl an der Anzahl der Iterationen ablesen kann. Diese *unäre* Darstellung von Zahlen ist zwar nicht so effizient wie eine Binärdarstellung, dafür aber deutlich weniger kompliziert.

Die wohl bekannteste dieser Repräsentationen wurde vom Mathematiker Alonzo Church entwickelt. Man nennt die entsprechenden λ-Terme daher auch *Church Numerals*. In dieser Repräsentation codiert man eine natürliche Zahl n durch einen λ-Term, der zwei Argumente benötigt und das erste insgesamt n-mal auf das zweite anwendet. Wir bezeichnen diese Darstellung einer Zahl n durch einen λ-Term mit $\overline{n}$. Auf diese Art wird eine Verwechselung der *Zahl* n mit ihrer Darstellung als *Term* vermieden.

Zur Vereinfachung der Beschreibung werden wir im folgenden die Abkürzung $f^n t$ für den λ-Term $\underbrace{f\ (f\ \ldots (f\ t)\ \ldots)}_{n\text{-mal}}$ verwenden.

Man beachte jedoch, daß $f^n t$ ist selbst kein korrekter λ-Term ist, sondern nur eine Kurzschreibweise, die eigentlich für jede konkrete Zahl separat definiert werden müßte.

Definition 18 (Church Numerals).
Das Church Numeral für eine Zahl $n \in \mathbb{N}$ ist der λ-Term $\overline{n} \equiv \lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^n\ \mathtt{x}$.

Die Repräsentation von natürlichen Zahlen durch λ-Terme ermöglicht es, für Funktionen auf natürlichen Zahlen einen Begriff der λ-Berechenbarkeit zu definieren und somit einen Vergleich zwischen dem λ-Kalkül und anderen Berechenbarkeitsmodellen zu ziehen. Die Definition ist naheliegend und hängt im Grunde auch nicht von der konkreten Repräsentation von Zahlen durch Church Numerals ab: eine Funktion f ist λ-berechenbar, wenn es einen λ-Term gibt, der sich auf der Repräsentation von Zahlen genauso verhält wie die Funktion f auf den entsprechenden Zahlen.[6]

Definition 19 (λ-Berechenbarkeit).
Eine (möglicherweise partielle) Funktion $f : \mathbb{N}^n \to \mathbb{N}$ heißt λ-berechenbar, wenn es einen λ-Term t gibt mit der Eigenschaft, daß für alle $x_1, \ldots, x_n, m \in \mathbb{N}$ gilt:

$$f(x_1, \ldots, x_n) = m \text{ genau dann, wenn } t\ \overline{x_1}\ \ldots \overline{x_n}\ = \overline{m}$$

Wir wollen im folgenden λ-Terme für die "Programmierung" der wichtigsten arithmetischen Operationen konstruieren. Diese "Programme" hängen natürlich von der konkreten Art ab, wie Zahlen durch λ-Terme dargestellt werden, denn sie müssen bei Anwendung auf die Repräsentation einer Zahl die Repräsentation des Operationsergebnisses liefern. Bei der Verwendung von Church-Numerals muß also die entsprechende Termvielfachheit bestimmt werden. So muß z.B. ein Term für die Nachfolgerfunktion bei Anwendung auf das Church-Numerals $\overline{n}$ immer den Term $\overline{n+1}$ und ein Term für die Addition bei Anwendung auf zwei Church-Numerals $\overline{m}$ und $\overline{n}$ immer den Term $\overline{m+n}$ liefern. Diese Überlegungen führen zu den folgenden Definitionen.

[6] Diese Definition von Berechenbarkeit gilt in ähnlicher Form für jedes Berechnungsmodell, da Computer immer nur auf Darstellungen von Zahlen arbeiten, nicht aber auf den Zahlen selbst.

Definition 20 (Arithmetische Operationen für Church Numerals).

$$
\begin{array}{lll}
\mathsf{s} & \equiv & \lambda \mathtt{n}.\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{n}\ \mathtt{f}\ (\mathtt{f}\ \mathtt{x}) \\
\mathsf{add} & \equiv & \lambda \mathtt{m}.\lambda \mathtt{n}.\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{m}\ \mathtt{f}\ (\mathtt{n}\ \mathtt{f}\ \mathtt{x}) \\
\mathsf{mul} & \equiv & \lambda \mathtt{m}.\lambda \mathtt{n}.\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{m}\ (\mathtt{n}\ \mathtt{f})\ \mathtt{x} \\
\mathsf{exp} & \equiv & \lambda \mathtt{m}.\lambda \mathtt{n}.\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{n}\ \mathtt{m}\ \mathtt{f}\ \mathtt{x} \\
\mathsf{zero} & \equiv & \lambda \mathtt{n}.\ \mathtt{n}\ (\lambda \mathtt{n}.\mathsf{F})\ \mathsf{T} \\
\mathsf{p} & \equiv & \lambda \mathtt{n}.\ (\mathtt{n}\ (\lambda \mathtt{fx}.\ (\mathsf{s}, \mathtt{let}\ (\mathtt{f},\mathtt{x}) = \mathtt{fx}\ \mathtt{in}\ \mathtt{f}\ \mathtt{x}))\ (\lambda \mathtt{z}.\overline{0}, \overline{0})).2 \\
\mathsf{PRs}[\mathit{base},h] & \equiv & \lambda \mathtt{n}.\ \mathtt{n}\ h\ \mathit{base}
\end{array}
$$

Die Terme s, add, mul, exp, p und zero repräsentieren die Nachfolgerfunktion, Addition, Multiplikation, Exponentiation, Vorgängerfunktion und einen Test auf Null. Der Term PRs[$base$,h] ist eine einfache Variante der primitiven Rekursion. Im Gegensatz zu den anderen Termen hat er (zugunsten einer klareren Display Form) zwei Parameter, die für die Basis- bzw. die Rekursionsfunktion stehen.

Bei der Repräsentation der *Nachfolgerfunktion* muß dafür gesorgt werden, daß bei Eingabe eines Terms $\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^n\,\mathtt{x}$ das erste Argument f einmal öfter als bisher auf das zweite Argument x angewandt wird. Dies wird dadurch erreicht, daß durch geschickte Manipulationen das zweite Argument durch (f x) ausgetauscht wird.

Bei der Simulation der *Addition* benutzt man die Tatsache, daß $\mathtt{f}^{m+n}\,\mathtt{x}$ identisch ist mit $\mathtt{f}^m\ (\mathtt{f}^n\,\mathtt{x})$ und ersetzt entsprechend in $\overline{m} \equiv \lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^m\,\mathtt{x}$ das zweite Argument x durch $\mathtt{f}^n\,\mathtt{x}$. Bei der *Multiplikation* muß man – entsprechend der Erkenntnis $\mathtt{f}^{m*n}\,\mathtt{x} \equiv (\mathtt{f}^m)^n\,\mathtt{x}$ – das erste Argument modifizieren und bei der *Exponentialfunktion* muß man noch einen Schritt weitergehen. Der *Test auf Null* ist einfach, da $\overline{0} \equiv \lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{x}$ ist. Angewandt auf $(\lambda \mathtt{n}.\mathsf{F})$ und T liefert dies T während jedes andere Church Numeral die Funktion $(\lambda \mathtt{n}.\mathsf{F})$ auswertet, also F liefert.

Die *Vorgängerfunktion* ist verhältnismäßig kompliziert zu programmieren, da wir bei Eingabe des Terms der Form $\overline{n}$=$\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^n\,\mathtt{x}$ eine Anwendung von f *entfernen* müssen. Dies geht nur dadurch, daß wir den Term komplett neu aufbauen. Wir tun dies, indem wir den Term $\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^n\,\mathtt{x}$ schrittweise abarbeiten und dabei die Nachfolgerfunktion s mit jeweils einem Schritt Verzögerung auf $\overline{0}$ anwenden. Bei der Programmierung müssen wir hierzu ein Paar $(\mathtt{f},\mathtt{x})$ verwalten, wobei x der aktuelle Ausgabewert ist und f die *im nächsten Schritt* anzuwendende Funktion. Startwert ist also $\overline{0}$ für x und $\lambda \mathtt{z}.\overline{0}$ für f, da im ersten Schritt ja ebenfalls $\overline{0}$ als Ergebnis benötigt wird. Ab dann wird für x immer der bisherige Wert benutzt und s für f.

Wir wollen am Beispiel der Nachfolgerfunktion und der Addition zeigen, daß sie tatsächlich durch die hier definierten λ-Terme berechnet werden.

Beispiel 21 *Es seien $m,n \in \mathbb{N}$ beliebige natürliche Zahlen.*

– *Wir zeigen, daß der Wert von* $\mathsf{s}\ \overline{n}$ *der Term* $\overline{n+1}$ *ist.*

$$
\begin{array}{lll}
\mathsf{s}\ \overline{n} & \equiv & (\lambda \mathtt{n}.\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{n}\ \mathtt{f}\ (\mathtt{f}\ \mathtt{x}))\ (\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^n\,\mathtt{x}) \\
 & \longrightarrow & \lambda \mathtt{f}.\lambda \mathtt{x}.\ (\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^n\,\mathtt{x})\ \mathtt{f}\ (\mathtt{f}\ \mathtt{x}) \\
 & \longrightarrow & \lambda \mathtt{f}.\lambda \mathtt{x}.\ (\lambda \mathtt{x}.\ \mathtt{f}^n\,\mathtt{x})\ (\mathtt{f}\ \mathtt{x}) \\
 & \longrightarrow & \lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^n\ (\mathtt{f}\ \mathtt{x}) \\
 & \longrightarrow & \lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^{n+1}\ \mathtt{x} \qquad\qquad \equiv \overline{n+1}
\end{array}
$$

– *Wir zeigen, daß* add $\overline{m}$ $\overline{n}$ *zu* $\overline{m+n}$ *reduziert.*

$$
\begin{aligned}
\mathsf{add}\ \overline{m}\ \overline{n} &\equiv (\lambda \mathtt{m}.\lambda \mathtt{n}.\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{m\ f\ (n\ f\ x)})\ \overline{m}\ \overline{n} \\
&\longrightarrow (\lambda \mathtt{n}.\lambda \mathtt{f}.\lambda \mathtt{x}.\ \overline{m}\ \mathtt{f\ (n\ f\ x)})\ \overline{n} \\
&\longrightarrow \lambda \mathtt{f}.\lambda \mathtt{x}.\ \overline{m}\ \mathtt{f\ (}\overline{n}\ \mathtt{f\ x)} \\
&\equiv \lambda \mathtt{f}.\lambda \mathtt{x}.\ (\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^m\,\mathtt{x})\ \mathtt{f\ (}\overline{n}\ \mathtt{f\ x)} \\
&\longrightarrow \lambda \mathtt{f}.\lambda \mathtt{x}.\ (\lambda \mathtt{x}.\ \mathtt{f}^m\,\mathtt{x})\ (\overline{n}\ \mathtt{f\ x)} \\
&\longrightarrow \lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^m\,(\overline{n}\ \mathtt{f\ x)} \\
&\equiv \lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^m\,((\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^n\,\mathtt{x})\ \mathtt{f\ x)} \\
&\longrightarrow \lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^m\,((\lambda \mathtt{x}.\ \mathtt{f}^n\,\mathtt{x})\ \mathtt{x)} \\
&\longrightarrow \lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^m\,(\mathtt{f}^n\,\mathtt{x}) \\
&\longrightarrow \lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^{m+n}\,\mathtt{x} \qquad\qquad \equiv\ \overline{m+n}
\end{aligned}
$$

Ähnlich leicht kann man zeigen, daß mul die Multiplikation, exp die Exponentiation und zero einen Test auf Null repräsentiert. Etwas schwieriger ist die Rechtfertigung der Vorgängerfunktion p. Die Rechtfertigung von PRs läßt sich durch Abwandlung der Listeninduktion aus Beispiel 23 erreichen.

4.4 Listen: Datencontainer

Endliche Listen dienen in der Programmierung als die einfachsten Container für eine unbestimmte Anzahl von Daten gleicher Natur. Mathematisch besehen sind sie eine einfache Erweiterung des Konzepts natürlicher Zahlen. Bei Zahlen startet man mit der Null und kann jede weitere Zahl dadurch bilden, daß man schrittweise die Nachfolgerfunktion s anwendet. Bei endlichen Listen startet man enstprechend mit einer leeren (null-elementige) Liste – bezeichnet durch [] – und bildet weitere Listen dadurch, daß man schrittweise Elemente a_i vor die bestehende Liste L anhängt. Die entsprechende Operation – bezeichnet durch $a_i.L$ – wird durch eine Funktion cons – das Gegenstück zur Nachfolgerfunktion s ausgeführt.

Definition 22 (Operatoren auf Listen).

$$
\begin{array}{lll}
[] & \equiv \mathsf{nil} & \equiv \lambda \mathtt{f}.\lambda \mathtt{x}.\mathtt{x} \\
t.list & \equiv \mathsf{cons}(t,list) & \equiv \lambda \mathtt{f}.\lambda \mathtt{x}.\,\mathtt{f}\ t\ (list\ \mathtt{f\ x}) \\
\mathsf{list\_ind}[base,h] & \equiv \mathsf{listind}(base,h) & \equiv \lambda \mathtt{list}.\ \mathtt{list}\ h\ base
\end{array}
$$

Die leere Liste ist also genauso definiert wie die Repräsentation der Zahl 0 während der Operator cons nun die Elemente der Liste – jeweils getrennt durch die Variable f – nebeneinanderstellt. Die Liste $a_1.a_2 \ldots a_n$ wird also dargestellt durch den Term

$$\lambda \mathtt{f}.\lambda \mathtt{x}.\,\mathtt{f}\ a_1\ (\mathtt{f}\ a_2\ \ldots(\mathtt{f}\ a_n\ \mathtt{x})\ \ldots)$$

Die Induktion auf Listen list_ind[$base$,h] ist eine Erweiterung der einfachen primitiven Rekursion. Ihr Verhalten beschreibt das folgende Beispiel.

Beispiel 23 *Es sei f definiert durch $f \equiv \mathsf{list\_ind}\,[base, h]$. Wir zeigen, daß f die Rekursionsgleichungen $f\ [] = base$ und $f\,(t.list) = h\ t\ (f\ list)$ erfüllt.*

Im Basisfall ist dies relativ einfach

$$\begin{aligned} f\ [] &\equiv \mathsf{list\_ind}\,[base, h]\ [] \equiv (\lambda \mathtt{list.}\ \mathtt{list}\ h\ base)\ [] \\ &\longrightarrow [\,]\ h\ base \qquad\qquad \equiv (\lambda \mathtt{f}.\lambda \mathtt{x}.\mathtt{x})\ h\ base \\ &\longrightarrow (\lambda \mathtt{x}.\mathtt{x})\ base \\ &\longrightarrow base \end{aligned}$$

Schwieriger wird es im Rekursionsfall:

$$\begin{aligned} f\,(t.list) &\equiv \mathsf{list\_ind}\,[base, h]\ t.list \equiv (\lambda \mathtt{list.}\ \mathtt{list}\ h\ base)\ t.list \\ &\longrightarrow t.list\ h\ base \qquad\qquad \equiv (\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}\ t\ (list\ \mathtt{f}\ \mathtt{x}))\ h\ base \\ &\longrightarrow (\lambda \mathtt{x}.\ h\ t\ (list\ h\ \mathtt{x}))\ base \\ &\longrightarrow h\ t\ (list\ h\ base) \end{aligned}$$

Dies ist offensichtlich nicht der gewünschte Term. Wir können jedoch zeigen, daß sich der Term $h\ t\ (f\ list)$ auf dasselbe Ergebnis reduzieren läßt.

$$\begin{aligned} h\ t\ (f\ list) &\equiv h\ t\ (\mathsf{list\_ind}\,[base, h]\ list) \\ &\equiv h\ t\ ((\lambda \mathtt{list.}\ \mathtt{list}\ h\ base)\ list) \\ &\longrightarrow h\ t\ (list\ h\ base) \end{aligned}$$

Die Rekursionsgleichungen von f beziehen sich also auf semantische Gleichheit und nicht etwa darauf, daß die linke Seite genau auf die rechte reduziert werden kann.

4.5 Programmierung rekursiver Funktionen

Die bisherigen Konstrukte erlauben uns, bei der Programmierung im λ-Kalkül Standardkonstrukte wie Zahlen, Tupel und Listen sowie bedingte Funktionsaufrufe zu verwenden. Uns fehlt nur noch eine Möglichkeit *rekursive Funktionsaufrufe* – das Gegenstück zur Schleife in imperativen Programmiersprachen – auf einfache Weise zu beschreiben. Wir benötigen also einen Operator, der es uns erlaubt, eine Funktion f durch eine rekursive Gleichung der Form

$$f\,(x) = t[f, x]^7$$

zu definieren, also durch eine Gleichung, in der f auf beiden Seiten vorkommt. Diese Gleichung an sich beschreibt aber noch keinen Term, sondern nur eine *Bedingung*, die ein Term zu erfüllen hat, und ist somit nicht unmittelbar für die Programmierung zu verwenden. Glücklicherweise gibt es jedoch ein allgemeines Verfahren, einen solchen Term direkt aus einer rekursiven Gleichung zu erzeugen. Wenn wir nämlich in der obigen Gleichung den Term t durch die Funktion $T \equiv \lambda \mathtt{f}.\lambda \mathtt{x}.t[\mathtt{f},\mathtt{x}]$ ersetzen, so können wir die Rekursionsgleichung umschreiben als $f\,(x) = T\ f\ x$ bzw. als

$$f = T\ f$$

Eine solche Gleichung zu lösen, bedeutet, einen *Fixpunkt* der Funktion T zu bestimmen, also ein Argument f, welches die Funktion T in eben dieses Argument selbst abbildet. Einen Operator R, welcher für beliebige Terme (d.h. also Funktionsgleichungen) deren Fixpunkt bestimmt, nennt man *Fixpunktkombinator* oder *Rekursor*.

[7] $t[f, x]$ ist ein Term, in dem die Variablen f und x frei vorkommen (siehe Def. 5 auf Seite 5).

Definition 24 (Fixpunktkombinator).
Ein Fixpunktkombinator ist ein λ-Term R mit der Eigenschaft, daß für jeden beliebigen λ-Term T die Gleichung $RT = t(RT)$ erfüllt ist.

Ein Fixpunktkombinator R liefert bei Eingabe eines beliebigen Terms T also eine Funktion $f \equiv RT$, für welche die rekursive Funktionsgleichung $f = T\,f$ erfüllt ist. Hat T nun die spezielle Gestalt $T \equiv \lambda\mathtt{f}.\lambda\mathtt{x}.t[\mathtt{f},\mathtt{x}]$, dann folgt $f(x) = (T\,f)\,x = t[f,x]$. Wenn wir also eine Funktion f durch die Gleichung $f(x) = t[f,x]$ spezifiziert haben, dann wissen wir, daß der λ-Term $R(\lambda\mathtt{f}.\lambda\mathtt{x}.t[\mathtt{f},\mathtt{x}])$ eine Lösung dieser Gleichung darstellt und wir können die reukursive Funktion f durch diesen Term "implementieren". In Anlehnung an die ML Notation bezeichnen wir diesen Term als letrec $f(x) = t[f,x]$, was den Gedanken wiederspiegelt, daß $f(x)$ durch die angegebene rekursive Gleichung definiert ist. Man beachte aber, daß der Ausdruck letrec $f(x) = t$ ein λ-Term ist und nicht etwa eine Deklaration, die den *Namen* f mit diesem Term verbindet.

Natürlich entsteht die Frage, ob Fixpunktkombinatoren überhaupt existieren. Für den λ-Kalkül kann diese Frage durch die Angabe konkreter Kombinatoren positiv beantwortet werden. Der bekannteste unter diesen ist der sogenannte Y-*Kombinator*.

Definition 25 (Der Fixpunktkombinator Y).

$$\begin{aligned} \mathsf{Y} &\equiv \lambda\mathtt{f}.(\lambda\mathtt{x}.\mathtt{f}\ (\mathtt{x}\ \mathtt{x}))(\lambda\mathtt{x}.\mathtt{f}\ (\mathtt{x}\ \mathtt{x})) \\ \text{letrec } f(x) = t &\equiv \mathsf{Y}(\lambda f.\lambda x.t) \end{aligned}$$

Wir zeigen nun, daß Y tatsächlich ein Fixpunktkombinator ist. Wie im Falle der Listeninduktion in Beispiel 23 ergibt sich die Gleichung $\mathsf{Y}(t) = t(\mathsf{Y}(t))$ allerdings nur dadurch, daß $\mathsf{Y}(t)$ und $t(\mathsf{Y}(t))$ auf denselben Term reduziert werden können.

$$\begin{aligned} \mathsf{Y}\ t &\equiv (\lambda\mathtt{f}.(\lambda\mathtt{x}.\mathtt{f}\,(\mathtt{x}\,\mathtt{x}))(\lambda\mathtt{x}.\mathtt{f}\,(\mathtt{x}\,\mathtt{x})))\ t \\ &\longrightarrow (\lambda\mathtt{x}.t\,(\mathtt{x}\,\mathtt{x}))\ (\lambda\mathtt{x}.t\,(\mathtt{x}\,\mathtt{x})) \\ &\longrightarrow t\ ((\lambda\mathtt{x}.t\,(\mathtt{x}\,\mathtt{x}))\ (\lambda\mathtt{x}.t\,(\mathtt{x}\,\mathtt{x}))) \end{aligned}$$

$$\begin{aligned} t\ (\mathsf{Y}\ t) &\equiv t\ ((\lambda\mathtt{f}.(\lambda\mathtt{x}.\mathtt{f}\,(\mathtt{x}\,\mathtt{x}))(\lambda\mathtt{x}.\mathtt{f}\,(\mathtt{x}\,\mathtt{x})))\ t) \\ &\longrightarrow t\ ((\lambda\mathtt{x}.t\,(\mathtt{x}\,\mathtt{x}))\ (\lambda\mathtt{x}.t\,(\mathtt{x}\,\mathtt{x}))) \end{aligned}$$

Ein Fixpunktkombinator, der sich auf seinen Fixpunkt reduzieren läßt, ist der Term $(\lambda\mathtt{x}.\lambda\mathtt{y}.\mathtt{y}\ (\mathtt{x}\ \mathtt{x}\ \mathtt{y}))\ (\lambda\mathtt{x}.\lambda\mathtt{y}.\mathtt{y}\ (\mathtt{x}\ \mathtt{x}\ \mathtt{y}))$.

5 Die Ausdruckskraft des λ-Kalküls

Zu Beginn dieser Abhandlung hatten wir die Behauptung aufgestellt, daß der λ-Kalkül das einfachste aller Modelle zur Erklärung von Berechenbarkeit sei. Wir wollen nun zeigen, daß der λ-Kalkül *Turing-mächtig* ist, also genau die Klasse aller berechenbaren Funktionen beschreiben kann.

Es ist leicht einzusehen, daß man alle λ-berechenbaren Funktionen programmieren kann. Für den Beweis der Gegenrichtung machen wir uns zunutze, daß Turingmaschinen und imperative Programmiersprachen in ihrer Berechnungskraft äquivalent zu den μ-rekursiven Funktionen sind. Es reicht daher zu zeigen, daß alle μ-rekursiven Funktionen λ-berechenbar sind, also daß jede μ-rekursive Funktion im λ-Kalkül simuliert werden kann. Wir fassen zu diesem Zweck die Definition der μ-rekursiven Funktionen kurz zusammen.

Definition 26 (μ-rekursive Funktionen).
Die Klasse der μ-rekursiven Funktionen ist induktiv wie folgt definiert.

1. *Die Nachfolgerfunktion s :$\mathbb{N}\to\mathbb{N}$ mit $s(x) = x+1$ für alle $x \in \mathbb{N}$ ist μ-rekursiv.*
2. *Die Projektionsfunktionen pr_k^n :$\mathbb{N}^n\to\mathbb{N}$ mit der Eigenschaft $pr_k^n(x_1,..,x_n) = x_k$ für alle $x_1,..,x_n \in \mathbb{N}$ sind μ-rekursiv für alle $n \in \mathbb{N}$ und $k \in \{1..n\}$.*
3. *Die Konstantenfunktionen c_k^n :$\mathbb{N}^n\to\mathbb{N}$ mit der Eigenschaft $c_k^n(x_1,..,x_n) = k$ für alle $x_1,..,x_n \in \mathbb{N}$ sind μ-rekursiv für alle $n, k \in \mathbb{N}$.*
4. *Die Komposition $f\circ(g_1,..,g_n)$:$\mathbb{N}^k\to\mathbb{N}$ der Funktionen f:$\mathbb{N}^n\to\mathbb{N}$, $g_1,..g_n$:$\mathbb{N}^k\to\mathbb{N}$ ist μ-rekursiv für alle $n, k \in \mathbb{N}$, wenn $f, g_1...g_n$ μ-rekursive Funktionen sind.*
   Dabei ist $f\circ(g_1,..,g_n)$ die eindeutig bestimmte Funktion h mit der Eigenschaft
   $h(\hat{x}) = f(\, g_1(\hat{x}),..,g_n(\hat{x})\,)$[8]
5. *Die primitive Rekursion $Pr[f,g]$:$\mathbb{N}^k\to\mathbb{N}$ zweier Funktionen f:$\mathbb{N}^{k-1}\to\mathbb{N}$ und g:$\mathbb{N}^{k+1}\to\mathbb{N}$ ist μ-rekursiv für alle $k \in \mathbb{N}$, wenn f und g μ-rekursiv sind.*
   Dabei ist $Pr[f,g]$ die eindeutig bestimmte Funktion h für die gilt
   $h(\hat{x},0) = f(\hat{x})$ *und*
   $h(\hat{x},y{+}1) = g(\hat{x},y,\, h(\hat{x},y))$.
6. *Die Minimierung μf :$\mathbb{N}^k\to\mathbb{N}$ einer Funktion f:$\mathbb{N}^{k+1}\to\mathbb{N}$ ist μ-rekursiv für alle $k \in \mathbb{N}$, wenn f μ-rekursiv ist.*
   Dabei ist μf die eindeutig bestimmte Funktion h, für die gilt

$$h(\hat{x}) = \begin{cases} min\{y \mid f(\hat{x},y){=}0\} & \textit{falls dies existiert und alle} \\ & f(\hat{x},i) \textit{ für } i{<}y \textit{ definiert sind} \\ \bot & \textit{sonst} \end{cases}$$

Mithilfe der λ-Terme, die wir im Abschnitt 4 definiert haben, ist es leicht, den λ-Kalkül als Turing-mächtig nachzuweisen.

Theorem 1. *Die Klasse der λ-berechenbaren Funktionen ist identisch mit der Klasse der μ-rekursiven Funktionen.*

Beweis: λ-berechenbare Funktionen lassen sich offensichtlich durch Programme einer der gängigen imperativen Programmiersprachen simulieren. Da diese sich wiederum durch Turingmaschinen beschreiben lassen und die Klasse der μ-rekursiven Funktionen mit der Klasse der Turing-berechenbaren Funktionen identisch ist, folgt hieraus, daß alle λ-berechenbaren Funktionen auch μ-rekursiv sind.

Wir können uns daher auf den Nachweis konzentrieren, daß man mit einem so einfachen Berechnungsmechanismus wie dem λ-Kalkül tatsächlich alle berechenbaren Funktionen repräsentieren kann. Wir weisen dazu nach, daß alle sieben Bedingungen der μ-rekursiven Funktionen aus Definition 26 durch λ-Terme erfüllt werden können.

1. Die Nachfolgerfunktion s haben wir in Beispiel 21 auf Seite 14 untersucht. Sie wird dargestellt durch den Term `s` $\equiv$ `λn.λf.λx. n f (f x)` und ist somit auch λ-berechenbar.

[8] $\hat{x}$ ist abkürzend für ein Tupel $(x_1,...,x_m)$

2. Die Projektionsfunktionen pr^n_m sind leicht darzustellen.
   Man wähle dazu den Term $\mathsf{pr}^n_m \equiv \lambda \mathtt{x}_1 . . \lambda \mathtt{x}_n . \mathtt{x}_m$.
3. Die Konstantenfunktionen c^n_m werden durch Terme dargestellt, die bei Eingabe von n Argumenten das Church-Numeral $\overline{\mathtt{m}}$ zurückgeben.
   Wir wählen $\mathsf{c}^n_m \equiv \lambda \mathtt{x}_1 . . \lambda \mathtt{x}_n . \overline{\mathtt{m}}$
4. Die Komposition $f \circ (g_1, .., g_n)$ läßt sich genauso leicht direkt nachbilden. Definieren wir $\circ_n \equiv \lambda \mathtt{f} . \lambda \mathtt{g}_1 . . \lambda \mathtt{g}_n . \lambda \mathtt{x} . \mathtt{f}\ (\mathtt{g}_1\ \mathtt{x}) . . (\mathtt{g}_n\ \mathtt{x})$, so simuliert $\circ_n$ den Kompositionsoperator für n-stellige Funktionen f.
   Denn für λ-berechenbare Funktionen $f, g_1 ... g_n$ und die zugehörigen λ-Terme $F, G_1 ... G_n$ repräsentiert $\circ_n \ F \ G_1 \ \ ... G_n$ die Komposition $f \circ (g_1, .., g_n)$, wie man durch Einsetzen leicht zeigen kann. Damit ist gezeigt, daß die Komposition λ-berechenbarer Funktionen wieder eine λ-berechenbare Funktion ist.
5. Mithilfe der Fixpunktkombinatoren läßt sich die primitive Rekursion zweier Funktionen auf einfache und natürliche Weise nachbilden. Wir müssen hierzu nur die Rekursionsgleichungen von $Pr[f, g]$ in eine einzige Gleichung umwandeln. Für $h = Pr[f, g]$ gilt
   $$h(\hat{x}, y) = \begin{cases} f(\hat{x}) & \text{falls } y = 0 \\ g(\hat{x}, y-1,\ h(\hat{x}, y-1)) & \text{sonst} \end{cases}$$
   Wenn wir die rechte Seite dieser Gleichung durch Conditional und Vorgängerfunktion beschreiben und anschließend den Y-Kombinator anwenden, erhalten wir eine Beschreibung der Funktion h durch einen λ-Term. Definieren wir also
   $$\mathsf{PR} \equiv \lambda \mathtt{f} . \lambda \mathtt{g} . \ \mathtt{letrec\ h(x)} = \lambda \mathtt{y} . \mathtt{if\ zero\ y\ then\ f\ x\ else\ g\ x\ (p\ y)\ (h\ x\ (p\ y))}$$
   so simuliert PR den Operator der primitiven Rekursion. Damit ist gezeigt, daß die primitive Rekursion μ-rekursiver Funktionen λ-berechenbar ist.
6. Auch die Minimierung μ-rekursiver Funktionen kann mit Fixpunktkombinatoren als λ-berechenbar nachgeweisen werden. Die Minimierung $\mu[f](\hat{x})$ ist im Endeffekt eine unbegrenzte Suche nach einem Wert y für den $f(\hat{x}, y) = 0$ ist. Startwert dieser Suche ist die Zahl 0.
   Die bei einem gegebenen Startwert y beginnende Suche nach einer Nullstelle von f läßt sich wie folgt durch eine rekursive Gleichung ausdrücken.
   $$\mathsf{min}_f(\hat{x}, y) = \begin{cases} y & \text{falls } f(\hat{x}, y) = 0 \\ \mathsf{min}_f(\hat{x}, y+1) & \text{sonst} \end{cases}$$
   Da f und $\hat{x}$ hierbei als Konstante aufgefaßt werden können, läßt sich die Gleichung vereinfachen, bevor wir den Y-Kombinator anwenden. Definieren wir
   $$\mathsf{Mu} \equiv \lambda \mathtt{f} . \lambda \mathtt{x} . \ (\mathtt{letrec\ min(y)} = \mathtt{if\ zero(f\ x\ y)\ then\ y\ else\ min\ (s\ y))}\ \overline{\mathtt{0}}$$
   so simuliert Mu den Minimierungsoperator μ. Damit ist gezeigt, daß die Minimierung μ-rekursiver Funktionen wieder λ-berechenbar ist.

Wir haben somit bewiesen, daß alle Grundfunktionen λ-berechenbar sind und die Klasse der λ-berechenbaren Funktionen abgeschlossen ist unter den Operationen Komposition, primitive Rekursion und Minimierung. Damit sind *alle μ-rekursiven Funktionen auch λ-berechenbar*. □

6 Semantische Fragen

Wir haben in den bisherigen Abschnitten die Syntax und die Auswertung von λ-Termen besprochen und gezeigt, daß man mit λ-Termen alle berechenbaren Funktionen ausdrücken kann. Offensichtlich ist auch, daß jeder λ-Term der Gestalt $\lambda x\,.\,t$ eine Funktion beschreiben muß. Jedoch ist unklar, mit welchem mathematischen Modell man eine solche Funktion als mengentheoretisches Objekt beschreiben kann.

Es ist nicht schwer, ein abstraktes Term-Modell für den λ-Kalkül anzugeben (siehe [2]): Jeder Term t beschreibt die Menge M_t der Terme, die semantisch gleich sind zu t (im Sinne von Definition 13). Jede Abstraktion $\lambda x\,.\,t$ repräsentiert eine Funktion $f_{\lambda x.t}$, welche eine Menge M_u in die Menge $M_{t[u/x]}$ abbildet. Diese Charakterisierung bringt uns jedoch nicht weiter, da sie keine Hinweise auf den Zusammenhang zwischen Funktionen des λ-Kalküls und 'gewöhnlichen' mathematischen Funktionen liefert.

Einfache mathematische Modelle, in denen λ-Terme als Funktionen eines festen Funktionenraumes interpretiert werden können, sind jedoch ebenfalls auszuschließen. Dies liegt daran, daß λ-Terme eine Doppelrolle spielen: Sie können als Funktion sowie als Argument einer anderen Funktion auftreten. Daher ist es möglich, λ-Funktionen zu konstruieren, die in sinnvoller Weise auf sich selbst angewandt werden können.

Beispiel 27 *Wir betrachten den Term* `twice` $\equiv$ `λf.λx.f (f x)`. *Angewandt auf zwei Terme* f *und* u *produziert dieser Term die zweifache Anwendung von* f *auf* u.

$$\texttt{twice}\ f\ u \xrightarrow{*} f\ (f\ u)$$

Damit ist `twice` *als Funktion zu verstehen. Andererseits kann* `twice` *auch auf sich selbst angewandt werden, wodurch eine Funktion entsteht, die ihr erstes Argument viermal auf das zweite anwendet:*

$$\begin{aligned}
&\texttt{twice twice} && \equiv \texttt{(λf.λx.f (f x)) twice}\\
\xrightarrow{*}\ &\texttt{λx. twice (twice x)} && \equiv \texttt{λx.(λf.λx.f (f x))(twice x)}\\
\xrightarrow{*}\ &\texttt{λx.λx'.(twice x)((twice x) x')} && \cong \texttt{λf.λx.(twice f)((twice f) x)}\\
\xrightarrow{*}\ &\texttt{λf.λx.(twice f) (f (f x))}\\
\xrightarrow{*}\ &\texttt{λf.λx.f (f (f (f x)))}
\end{aligned}$$

Die übliche mengentheoretische Sicht von Funktionen muß die Selbstanwendung von Funktionen jedoch verbieten. Sie identifiziert nämlich eine Funktion f mit ihrem Graphen, d.h. der Menge $\{(x, y) \mid y{=}f(x)\}$. Wäre eine selbstanwendbare Funktion wie `twice` mengentheoretisch interpretierbar, dann müsste `twice` eine Menge sein, die für ein y das Element $(\texttt{twice}, y)$ enthält. Dies aber verletzt ein fundamentales Axiom der Mengentheorie, welches fordert, daß eine Menge sich selbst nicht enthalten darf.[9] Wir können also nicht erwarten, 'natürliche' Modelle für den λ-Kalkül zu finden. Dies wird nur möglich sein, wenn wir die erlaubten Terme auf syntaktische Art einschränken.[10] Diese Problematik betrifft nicht nur den λ-Kalkül, sondern *alle* Berechenbarkeitsmodelle, da diese, wie in Abschnitt 5 gezeigt, äquivalent zum λ-Kalkül sind.

[9] Ein Verzicht auf dieses Axiom würde zum *Russelschen Paradox* führen: Man betrachte die Menge $M = \{X \mid X \notin X\}$ und untersuche, ob $M \in M$ ist oder nicht. Wenn wir $M \in M$ annehmen, so folgt nach Definition von M, daß M – wie jedes andere Element von M – nicht in sich selbst enthalten ist, also $M \notin M$. Da M aber die Menge *aller* Mengen ist, die sich selbst nicht enthalten, muß M ein Element von M sein: $M \in M$.

Ein mathematisches Modell für berechenbare Funktionen ist die *Domain-Theorie*, die Anfang der siebziger Jahre von Dana Scott [5,6] entwickelt wurde. Diese Theorie basiert auf topologischen Konzepten wie Stetigkeit und Verbänden. Sie benötigt jedoch ein tiefes Verständnis komplexer mathematischer Theorien. Aus diesem Grunde werden wir auf die *denotationelle Semantik* des λ-Kalküls nicht weiter eingehen.

7 Eigenschaften des λ-Kalküls

Bei den bisherigen Betrachtungen sind wir stillschweigend davon ausgegangen, daß jeder λ-Term einen Wert besitzt, den wir durch Reduktion bestimmen können. Wie weit aber müssen wir gehen, bevor wir den Prozeß der Reduktion als beendet erklären können? Die Antwort ist eigentlich naheliegend: wir hören erst dann auf, wenn nichts mehr zu reduzieren ist, also der entstandene Term kein Redex im Sinne der Definition 10 enthält. Diese Überlegung führt dazu, eine Teilklasse von λ-Termen besonders hervorzuheben, nämlich solche, die nicht mehr reduzierbar sind. Diese Terme repräsentieren die Werte, die als Resultat von Berechnungen entstehen können.

Definition 28 (Normalform). *Es seien s und t beliebige λ-Terme.*

1. *t ist in Normalform, wenn t keine Redizes enthält.*
2. *t ist normalisierbar, wenn es einen λ-Term in Normalform gibt, auf den t reduziert werden kann.*
3. *t heißt Normalform von s, wenn t in Normalform ist und $s \xrightarrow{*} t$ gilt.*

Diese Definition wirft eine Reihe von Fragen auf, die wir im folgenden diskutieren.

7.1 Hat jeder λ-Term eine Normalform?

In Anbetracht der Tatsache, daß der λ-Kalkül Turing-mächtig ist, muß diese Frage natürlich mit *nein* beantwortet werden. Bekanntermaßen enthalten die rekursiven Funktionen auch die partiellen Funktionen – also Funktionen, die nicht auf allen Eingaben definiert sind – und es ist nicht möglich, einen Formalismus so einzuschränken, daß nur totale Funktionen betrachtet werden, ohne daß dabei gleichzeitig manche berechenbare totale Funktion nicht mehr beschreibbar ist.[11] Diese Tatsache muß sich natürlich auch auf die Reduzierbarkeit von λ-Termen auswirken: es gibt Reduktionsketten, die nicht terminieren. Wir wollen es jedoch nicht bei dieser allgemeinen Antwort belassen, sondern einen konkreten Term angeben, der nicht normalisierbar ist.

Lemma 1. *Der Term* `(λx. x x) (λx. x x)` *besitzt keine Normalform.*

Beweis: Bei der Reduktion von `(λx. x x) (λx. x x)` gibt es genau eine Möglichkeit. Wenn wir diese ausführen, erhalten wir denselben Term wie zuvor und können den Term somit unendlich lange weiterreduzieren. □

[10] Ein einfaches aber wirksames Mittel ist hierbei die Forderung nach *Typisierbarkeit* der Terme. Die Zuordnung von Typen zu Termen beschreibt bereits auf syntaktischem Wege, zu welcher Art von Funktionenraum eine Funktion gehören soll.

[11] Wie wir in einer späteren Einheit zeigen werden, ist die Menge der total-rekursiven Funktionen nicht rekursiv-aufzählbar.

7.2 Führt jede Reduktionsfolge zu einer Normalform, wenn ein λ-Term normalisierbar ist?

Im Beispiel 12 (Seite 8) hatten wir bereits festgestellt, daß es unter Umständen mehrere Möglichkeiten gibt, einen gegebenen λ-Term zu reduzieren. Auch wissen wir schon, daß nicht jede Reduktionskette terminieren muß. So stellt sich natürlich die Frage, ob es etwa Terme gibt, bei denen eine Strategie, den zu reduzierenden Teilterm auszuwählen, zu einer Normalform führt, während eine andere zu einer nichtterminierenden Reduktionsfolge führt. Dies ist in der Tat der Fall.

Lemma 2. *Es gibt normalisierbare λ-Terme, bei denen nicht jede Reduktionsfolge zu einer Normalform führt.*

Beweis: Es seien $\mathsf{W} \equiv \lambda\mathtt{x.\,x\ x\ x}$, $\mathsf{I} \equiv \lambda\mathtt{x.\,x}$ und $t \equiv (\lambda\mathtt{x.}\lambda\mathtt{y.\,y})\,(\mathsf{WW})\,\mathsf{I}$.
Es gibt zwei Möglichkeiten, den Term t zu reduzieren. Wählen wir die am weitesten links-stehende, so ergibt sich als Reduktionsfolge:

$$(\lambda\mathtt{x.}\lambda\mathtt{y.\,y})(\mathsf{WW})\mathsf{I} \xrightarrow{*} (\lambda\mathtt{y.\,y})\ \mathsf{I} \xrightarrow{*} \mathsf{I}$$

womit wir eine Normalform erreicht hätten. Wenn wir aber den Teilterm (WW) zuerst reduzieren, dann erhalten wir (WWW). Reduzieren wir dann wieder im gleichen Bereich, so erhalten wir folgende Reduktionskette

$$\begin{aligned}(\lambda\mathtt{x.}\lambda\mathtt{y.\,y})(\mathsf{WW})\mathsf{I} &\xrightarrow{*} (\lambda\mathtt{x.}\lambda\mathtt{y.\,y})(\mathsf{WWW})\mathsf{I} \\ &\xrightarrow{*} (\lambda\mathtt{x.}\lambda\mathtt{y.\,y})(\mathsf{WWWW})\mathsf{I} \xrightarrow{*} \ldots\end{aligned}$$

Diese Kette erreicht niemals eine Normalform. □

7.3 Wie kann man eine Normalform finden, wenn es eine gibt?

Wir wissen nun, daß die Bestimmung einer Normalform von der Reduktionsstrategie abhängen kann. Daraus ergibt sich unmittelbar die Frage, ob es denn wenigstens eine einheitliche Strategie gibt, mit der man eine Normalform finden kann, wenn es sie gibt? Die Beantwortung dieser Frage ist von fundamentaler Bedeutung für die praktische Verwendbarkeit des λ-Kalküls als Programmiersprache. Ohne eine Reduktionsstrategie, mit der man garantiert eine Normalform finden kann, wäre der λ-Kalkül als Grundlage der Programmierung unbrauchbar. Glücklicherweise kann man diese Frage positiv beantworten

Lemma 3 (Leftmost-Reduktion). *Reduziert man in einem λ-Term immer das jeweils am weitesten links stehende (äußerste) Redex, so wird man eine Normalform finden, wenn der Term normalisierbar ist.*

Intuitiv läßt sich der Erfolg dieser Strategie wie folgt begründen. Der Beweis von Lemma 2 hat gezeigt, daß normalisierbare Terme durchaus Teilterme enthalten können, die selbst nicht normalisierbar sind. Diese Teilterme können nun zur Bestimmung der Normalform nichts beitragen, da ihr Wert ja nicht festgestellt werden kann. Wenn wir daher die äußerste Funktionsanwendung zuerst reduzieren, werden wir feststellen, ob ein Teilterm überhaupt benötigt wird, bevor wir ihn unnötigerweise reduzieren.

Die Strategie, die Funktionsargumente zuerst einzusetzen, bevor ihr Wert bestimmt wird, entspricht der *call-by-name* Auswertung in Programmiersprachen. Sie ist die sicherste Reduktionsstrategie, aber die Sicherheit wird oft auf Kosten der Effizienz erkauft. Es mag nämlich sein, daß durch die Reduktion ein Argument verdoppelt wird und daß wir es somit zweimal reduzieren müssen. Eine *call-by-value* Strategie hätte uns diese Doppelarbeit erspart, aber diese können wir nur anwenden, wenn wir wissen, daß sie garantiert terminiert. Da das Halteproblem jedoch unentscheidbar ist, gibt es leider keine Möglichkeit, dies für beliebige λ-Terme im Voraus zu entscheiden. Ein präziser Beweis für diese Aussage ist verhältnismäßig aufwendig. Wir verweisen daher auf Lehrbücher über den λ-Kalkül wie [1,7] für Details.

7.4 Ist die Normalform eines λ-Terms eindeutig?

Auch hierauf benötigen wir eine positive Antwort, wenn wir den λ-Kalkül als Programmiersprache verwenden wollen. Wenn nämlich bei der Auswertung eines λ-Terms verschiedene Reduktionsstrategien zu verschiedenen Ergebnissen (Normalformen) führen würden, dann könnte man den Kalkül zum Rechnen nicht gebrauchen, da nicht eindeutig genug festgelegt wäre, was der Wert eines Ausdrucks ist.

Rein syntaktisch betrachtet ist der Reduktionsmechanismus des λ-Kalküls ein *Termersetzungssystem*, welches Vorschriften angibt, wie Terme in andere Terme umgeschrieben werden dürfen (engl. *rewriting*). In der Denkweise der Termersetzung ist die Frage nach der Eindeutigkeit der Normalform ein Spezialfall der Frage nach der *Konfluenz* eines Regelsystems, also der Frage, ob zwei Termersetzungsketten, die im gleichen Term begonnen haben, wieder zusammengeführt werden können.

Konfluenzbeweise sind verhältnismäßig einfach, wenn man weiß, daß ein Termersetzungssystem konfluent ist. Für den uneingeschränkten λ-Kalkül ist der Beweis des Konfluenztheorems, das nach den Mathematikern A. Church und B. Rosser benannt wurde, allerdings relativ aufwendig. Für Details verweisen wir daher wiederum auf Lehrbücher wie [1,3,7].

Theorem 2 (Church-Rosser Theorem).
Es seien t, u und v beliebige λ-Terme und es gelte $t \xrightarrow{*} u$ und $t \xrightarrow{*} v$. Dann gibt es einen λ-Term s mit der Eigenschaft $u \xrightarrow{*} s$ und $v \xrightarrow{*} s$.

Eine unmittelbare Konsequenz dieses Theorems ist, daß die Normalformen eines gegebenen λ-Terms bis auf α-Konversionen eindeutig bestimmt sind.

Korollar 29 *Es seien t, u und v beliebige λ-Terme.*

1. *Alle Normalformen von t sind kongruent im Sinne von Definition 9.*
2. *Gilt u=v, dann gibt es einen λ-Term s mit der Eigenschaft $u \xrightarrow{*} s$ und $v \xrightarrow{*} s$.*
3. *Gilt u=v und v ist in Normalform, so folgt $u \xrightarrow{*} v$.*
4. *Gilt u=v, so haben u und v dieselben Normalformen oder überhaupt keine.*
5. *Sind u und v in Normalform, dann sind sie entweder kongruent oder nicht gleich.*

Es ist also legitim, λ-Terme als Funktionen anzusehen und deshalb ist es auch möglich, formale Methoden für Beweise über Berechnungen mit λ-Termen einzusetzen. Dies ist allerdings ein Thema, das den Rahmen einer Einführungsveranstaltung sprengt.

Literatur

1. Henk P. Barendregt. *The Lambda Calculus. Its syntax and semantics*, volume 103 of *Studies in Logic and The Foundations of Mathematics*. North–Holland, 1981. 23
2. Alonzo Church and B. J. Rosser. Some properties of conversion. *Trans. Am. Math. Soc.*, 39:472–482, 1936. 20
3. J. Roger Hindley and Jonathan P. Seldin. *Introduction to combinators and λ-calculus*. Cambridge University Press, 1986. 23
4. Hartley jr. Rogers. *Theory of recursive functions and effective computability*. MIT Press, Cambridge, MA, 1967.
5. Dana Scott. Lattice theory, data types, and semantics. In R. Rustin, editor, *Formal semantics of Programming Languages*, pages 65–106. Prentice Hall, 1972. 21
6. Dana Scott. Data types as lattices. *SIAM Journal on Computing*, 5:522–287, 1976. 21
7. Sören Stenlund. *Combinators, λ-terms and Proof Theory*. D. Reidel, Dordrecht, The Netherlands, 1972. 23