Fachbereich 6.2 - Informatik
Naturwissenschaftlich-Technische Fakultät I
Universität des Saarlandes

Cubint

Ein Interpreter auf Basis des λ-Cubes

Christian Müller
cmueller@ps.uni-sb.de
Programming System Lab

Fortgeschrittenen-Praktikum

Angefertigt unter Leitung von
Prof. Dr. Gert Smolka

Betreut von
Guido Tack und Gert Smolka

16. November 2004

Inhaltsverzeichnis

1 Einleitung

Der λ-Cube [1] ist eine uniforme Theorie, die einer Reihe getypter λ-Kalküle einen gemeinsamen formalen Rahmen gibt. Kennzeichnend ist, daß keine Unterscheidung zwischen Typen und Termen vorgenommen wird; es gibt nur Ausdrücke. Der Interpreter *Cubint* basiert auf dieser Theorie und beinhaltet zudem typische Konstrukte aus Programmiersprachen. So ist es möglich mit verschiedenen getypten λ-Kalkülen, zum Beispiel dem einfach getypten λ-Kalkül, System F, F_ω oder dem Calculus of Conststructions, in einer gemeinsamen Umgebung zu experimentieren. Mit entsprechenden Kommandos kann man *on-the-fly* den Modus des Type-Checkers wechseln, sich ein Ausführungsprotokoll anzeigen lassen oder das Format der Ausgabe des Interpreters verändern. *Cubint* stellt somit eine nützliches Tool zur Vertiefung des Verständnisses beim Studium von Typsystemen dar.

Zuerst wird die Syntax des λ-Cubes vorgestellt und einige Sprechweisen geklärt. Auf der Syntax wird der Auswertungsbegriff definiert. Danach wird der Type-Checker von *Cubint* behandelt. Schwerpunkt ist hier die Beschreibung, wie die Erweiterungen in den λ-Cube integriert wurden, so daß sie sich in den Kalkülen S, F und F_ω in gewohnter Art und Weise verwenden lassen. In Abschnitt 5 wird Typrekonstruktion auf einem eingeschränkten Kalkül beschrieben und Unterschiede und Parallelen zu ML aufgezeigt.
Abschnitt 7 behandelt die Architektur des Interpreters. Das Zusammenspiel der Komponenten wird anhand einer Strukturzeichnung erläutert. Anschließend wird auf die wesentlichen Aspekte der einzelnen Einheiten eingegangen. Am Schluß wird kurz ein Szenario zum Einsatz von *Cubint* in der Lehre vorgestellt.

2 Die interne Syntax von *Cubint*

Die Besonderheit des λ-Cubes ist seine uniforme Syntax. Es gibt keine syntaktische Unterscheidung zwischen Termen, Typen und Kinds. Diese Eigenschaft ist aus vielerlei Hinsicht attraktiv. Die Darstellung ist kompakt und verdeutlicht die Gemeinsamkeiten der einzelnen Konstrukte. Auf der theoretischen Seite führt dies zu einem kompakten Formalismus, was für Beweise sehr nützlich sein kann. Aus praktischer Sicht (speziell für *Cubint*) erreicht man sehr schlanke Algorithmen.

Abbildung 1 stellt die Syntax des erweiterten λ-Cubes dar. Im folgenden werden wir diesen Kalkül LC^x nennen.

Das Konstrukt Π subsumiert die sonst üblichen Typschreibweisen für Funktionstypen.

- $T_1 \rightarrow T_2$ ist eine abkürzende Schreibweise für $\Pi x : T_1.T_2 \quad x \notin T_2$. Die Variable x darf nicht im Typ T_2 vorkommen.
- $\forall X : K.T$ kann in der neuen Syntax als $\Pi X : K.T$ geschrieben werden.

Die Implementierung verwendet de Bruijn-Indizes zur Darstellung gebundener Namen. Aufgrund der besseren Lesbarkeit verzichten wir auf die de Bruijn-Schreibweise, da die meisten Aspekte mit expliziten Namen vollkommen analog

E	::=	x	Variable
	|	$\star$	Kind
	|	$\Box$	Box
	|	$E\,E$	Applikation
	|	$\lambda x : E.E$	Abstraktion
	|	$\Pi x : E.E$	Produkt
	|	base types	
	|	consts	
	|	$E\,binop\,E$	
	|	$uop\,E$	
	|	if E then E else E	Konditional
	|	E as E	Zusicherung
	|	$\mu X.E$	rekursiver Typ
	|	fold $E\,E$	Falten iso-rek. Typen
	|	unfold $E\,E$	Auffalten iso-rek. Typen
	|	$\{l_i = E^{\,i\in 1..n}\}$	Record
	|	$\{l_i : E^{\,i\in 1..n}\}$	Record Typ
	|	$E.l$	Projektion
	|	$< l = E >$ as E	Variante
	|	$< l_i : E^{\,i\in 1..n} >$	Varianten Typ
	|	case E of $< l_i = x_i > \Rightarrow E^{i\in 1..n}$	

basetype	::=	Int | Bool | Unit
uop	::=	$\sim$ | not
binop	::=	$<> \mid <= \mid >= \mid = \mid + \mid - \mid *$
consts	::=	true | false | unit | n
n	::=	$0 \mid \ldots \mid 9 \mid n\,0 \mid \ldots \mid n\,9$

Abbildung 1: Syntax von LC^x

behandelt werden können. An den Stellen, an welchen die Analogie nicht gegeben ist, wird explizit darauf hingewiesen.

2.1 Sprechweisen

Die syntaktischen Einheiten von LC^x nennt man *Ausdrücke*. Sie lassen sich in vier Kategorien einteilen: Terme, Typen, Kinds und $\Box$, welche man zusammenfassend als *Sorten* bezeichnet. Die Sorten bilden eine Hierarchie, wobei $Term < Typ < Kind < \Box$ gilt. $\Box$ schließt die Hierarchie also nach oben hin ab. Der Begriff *Typ* wird in den kommenden Kapiteln in zwei verschiedenen Bedeutungen verwendet:

1. als Bezeichner der Sorte *Typ*
2. als Resultat der Typüberprüfung. Die Zahl 3 hat den Typ *Int*, und der Typ *Int* hat den Typ $\star$.

Wenn man über einen Ausdruck des Typs A spricht, kann der Ausdruck von der Sorte Term, Typ oder Kind sein. Aus dem Kontext wird hervorgehen auf welche Bedeutung wir uns beziehen.
Wenn über den *Basiskalkül* gesprochen wird, beziehen wir uns auf die ersten sechs Konstrukte von LC^x, also von „Variable“ bis „Produkt“.

3 Auswertung von Ausdrücken

Die Syntax ist isoliert gesehen nicht sonderlich spannend. In diesem Abschnitt soll ihr Leben eingehaucht werden. Hierfür benötigen wir Regeln zur syntaktischen Umformung von Ausdrücken. *Cubint* verwendet zur Auswertung von Ausdrücken eine Einschritt-Semantik mit schwacher β-Reduktion. Wir beschränken uns auf die Darstellung der operationalen Semantik für den Basiskalkül. Die Regeln für die Erweiterungen entsprechen im wesentlichen den Regeln aus dem Buch *Types and Programming Languages* von Pierce [11].

$$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad \text{(Eval-App1)}$$

$$\frac{e_2 \to e_2'}{v_1\ e_2 \to v_1\ t_2'} \qquad \text{(Eval-App2)}$$

$$(\lambda x : T_{11}.e_{12})\, v_2 \to [x \mapsto v_2]e_{12} \qquad \text{(Eval-App3)}$$

Kleinbuchstaben e_i und e_i' stehen für beliebige Ausdrücke aus LC^x. Der Buchstabe v bezeichnen eine besondere Klasse, die man *Werte* nennt. *Werte* sind Ausdrücke, die nicht mit den gegebenen Regeln umgeformt werden können. Die Menge der Werte ist definiert als:

$$\begin{array}{rcl} V & ::= & x \\ & | & \mathit{consts} \\ & | & \lambda x : E_1.E_2 \\ & | & < l = V > \text{ as } T \\ & | & \{l_i = V_i\ ^{i \in 1..n}\} \\ & | & \text{fold } T\ V \\ & | & \text{Int} \mid \text{Bool} \mid \text{Unit} \\ & | & \Pi x : E_1.E_2 \\ & | & \mu X.E \\ & | & \{l_i : E_i\ ^{i \in 1..n}\} \\ & | & < l_i : E_i\ ^{i \in 1..n} > \end{array}$$

Umformung beziehungsweise Auswertung ist wiefolgt definiert:

Definition (Auswertung):
Die Auswertungsrelation $\rightarrow_\beta$ enthält alle Paare, die über die Regeln Eval-App1-3 erzeugt werden können. $\rightarrow_\beta^\star$ bezeichnet den reflexiven, transitiven Abschluß der Relation.

Die Relation kann ohne Schwierigkeiten auf LC^x erweitert werden. Die Tatsache, daß der Ausdruck $(\lambda x : Int.x + 1)\ 3$ zu 4 auswertet, kann formal als $(\lambda x : Int.x + 1)\ 3 \rightarrow_\beta^\star 4$ geschrieben werden. Anstelle von Auswertung spricht man auch oft von Reduktion.
Ein Ausdruck ist in *Normalform*, wenn er nicht mehr weiter reduziert werden kann. Wie wir in Abschnitt 3.3 sehen werden, deckt sich die Menge der Ausdrücke in Normalform nicht mit der Menge der Werte.

3.1 Alternative Auswertungsstrategie

Die β-Reduktion mit Substitution ist theoretisch motiviert. Für die Implementierung eines Programmiersystems ist sie aus Effizienzgründen ungeeignet. In der Regel Eval-App3 muß der Rumpf e_{12} der λ-Abstraktion nach Vorkommen der Variablen x durchsucht werden. Jedes Vorkommen von x wird durch den Wert v_2 ersetzt, wodurch man oftmals viele Kopien von v_2 erzeugen muß. Es ist deutlich effizienter, die Evaluation mit einer Umgebung zu realisieren. Die Regel Eval-App3 kann dann in der folgenden Form geschrieben werden:

$$(\lambda x : T_{11}.e_{12})\ v_2|\Sigma \rightarrow e_{12}|\Sigma, x \mapsto v_2$$

Es wird also keine Substitution vorgenommen, sondern einfach die Umgebung erweitert. Wenn man auf eine freie Variable stößt, schaut man nach, ob für sie ein Ausdruck in der Umgebung gespeichert ist. Folglich muß keine Kopie des Wertes erzeugt werden. Da dieses Modell deutlich von der theoretischen Betrachtung abweicht, wurde in *Cubint* die weniger effiziente Variante gewählt.

3.2 Auswertung unterhalb von Bindern

Die Ausdrücke $\lambda x : E_1.E_2$ und $\Pi x : E_1.E_2$ binden die Variable x in E_2. Bei der schwachen β-Reduktion wird unterhalb von Bindern nicht reduziert. Deshalb scheint es auf den ersten Blick etwas seltsam zu sein, daß Variablen als Werte ausgezeichnet sind. Diese Festlegung ist aber für die höherstufigen Kalküle F_ω und CC in *Cubint* notwendig, da hier bei der Typüberprüfung an bestimmten Stellen der Evaluator aufgerufen werden muß. Schauen wir uns zur Verdeutlichung ein Beispiel an:

$$\begin{aligned} F &:= \lambda X.\, X \rightarrow X \\ e &:= \lambda X.\lambda f : \mathbf{F}\ \mathbf{X}.\lambda x : X.\, f\ x \end{aligned}$$

Damit die Applikation $f\ x$ im Rumpf von e Sinn macht, muß f einen Typ der Form $X \rightarrow T$ haben. Zur Überprüfung muß $F\ X$ ausgewertet werden. Der Evaluator stößt hierbei auf die abstrakte Variable X, welche er per Definition als Wert ansieht. Somit kann die Regel Eval-App3 angewendet werden.
In *Cubint* funktioniert das Auswerten unterhalb von Bindern also ohne technischen Mehraufwand. Anders sieht es beim Umgebungsmodell aus Abschnitt

3.1 aus. Der Evaluator muß nämlich zwei Arten von Variablen unterscheiden können: diejenigen, für die ein Wert in der Umgebung vermerkt ist, und die gebunden. Aus technischer Sicht erfordert die Verbindung von Typüberprüfung und Auswertung eine kompliziertere Verwaltung der Umgebung. Sowohl der Type-Checker als auch der Evaluator arbeiten mit Umgebungen zur Speicherung von Kontextinformationen. Hierbei ist in der Implementierung darauf zu achten, daß bei beiden die de Bruijn-Indizes konsistent bleiben. Dies ist ein weiterer Grund, weshalb in *Cubint* das theoretische Modell bevorzugt wurde.

3.3 Auswertungsfehler

Ausgestattet mit einer Semantik können in LC^x schon sehr viele sinnvolle Strukturen realisiert werden; beispielsweise das Church-Encoding natürlicher Zahlen oder abstrakte Datentypen. Ein böswilliger Benutzer kann allerdings die Auswertung sehr leicht zum Stocken bringen. Die Syntax enthält viele Konstrukte, die nur für eine ganz bestimmte Art von Ausdrücken Sinn machen. Der Ausdruck $4 + \mathit{true}$ sollte zum Beispiel als fehlerhaft zurückgewiesen werden. Er ist nämlich weder ein Wert, noch kann er weiter reduziert werden. Solche Terme bezeichnet man als *stuck terms*. Die Menge der Werte V ist also nicht identisch zur Menge der Ausdrücke in Normalform E_{nf}. Wenn S die Menge der *stuck terms* bezeichnet, so erhalten wir $E_{\mathrm{nf}} = V \uplus S$. Ziel des nächsten Abschnittes ist es, die Ausdrücke aus S herauszufiltern, bevor sie die Auswertung erreichen. Dann liefert uns die Reduktion auf Normalform nämlich die gewünschte Menge V.

4 Typüberprüfung von LC^x

Mit einem Typsystem kann man Ausdrücke auf Wohlgetyptheit überprüfen. *Wohlgetyptheit* bedeutet, daß kein Ausdruck, der von der Typüberprüfung akzeptiert wird, zu einem *stuck term* auswertet. Im λ-Cube fällt der Typüberprüfung eine weitere zentrale Aufgabe zu. Von ihr wird die Kategorisierung in Sorten vorgenommen, das heißt jedem Ausdruck wird eindeutig eine Sorte zugeordnet.

Wir steigen ein mit der Definition einer Äquivalenzrelation auf Ausdrücken, die in vielen Typregeln benötigt wird. Danach werden die Inferenzregeln für den Basiskalkül entwickelt. In diesem Zusammenhang werden Parallelen zu der konventionellen Darstellung getypter λ-Kalküle gezogen. Das Basissystem wird Schritt für Schritt auf die volle Syntax von LC^x erweitert. Hierbei wird nicht immer die allgemeinste Form verwendet, sondern darauf geachtet, daß sich die Erweiterungen in S, F und F_ω wie in Programmiersprachen üblich benutzen lassen.

4.1 Äquivalenz auf Ausdrücken – die Relation $\equiv_\beta$

Die Applikation $(\lambda x : ((\lambda Y.Y)Int).x + 1)\ 3$ ist wohlgetypt, wenn wir der Typ des Formalparameters mit dem Typ des Aktualparameters übereinstimmt. Diese Überprüfung kann mit der Relation $\equiv_\beta$ durchgeführt werden. Sie testet, ob zwei Ausdrücke modulo schwacher β-Reduktion gleich sind. Da $(\lambda Y.Y)Int \equiv_\beta Int$ gilt, ist die obige Applikation wohlgetypt.

Der Test auf syntaktische Gleichheit reicht in *Cubint* aus zwei Gründen nicht aus:

1. Die syntaktischen Konstrukte sind in der Implementierung mit zusätzlichen Informationen ausgestattet. Unterscheiden sich zum Beispiel gebundene Namen, so gibt der Test auf syntaktische Gleichheit aus, daß die Ausdrücke nicht äquivalent sind.

2. In den Kalkülen F_ω und CC gibt es *Typoperatoren*. Um die vom Benutzer eingeführte Typdefinitionen in der erwarteten Form ausgegeben zu können, verzichtet *Cubint* auf eine globale Normalform-Reduktion von Typen. Die Annotation in $\lambda x : Id\ Int.x$ bleibt somit in der Ausgabe des Interpreters erhalten. Der Äquivalenztest nimmt die Auswertung in der Regel E-RED vor.

Für die Kalküle S, F und F_ω reichen die folgenden Regeln aus:

$$\frac{T \rightarrow_\beta^\star S \qquad T' \rightarrow_\beta^\star S' \qquad S \equiv_\beta S'}{T \equiv_\beta T'} \tag{E-RED}$$

$$\frac{T = T'}{T \equiv_\beta T'} \tag{E-REFL}$$

$$\frac{E_1 \equiv_\beta E_1' \qquad E_2 \equiv_\beta E_2'}{\lambda x : E_1.E_2 \equiv_\beta \lambda y : E_1'.E_2'} \tag{E-LAM}$$

$$\frac{E_1 \equiv_\beta E_1' \qquad E_2 \equiv_\beta E_2'}{\Pi x : E_1.E_2 \equiv_\beta \Pi y : E_1'.E_2'} \tag{E-PI}$$

$$\frac{E \equiv_\beta E'}{\mu X.E \equiv_\beta \mu Y.E'} \tag{E-MU}$$

$$\frac{l_i = l_i' \qquad E_i \equiv_\beta E_i'}{\{l_i = {E_i}^{i \in 1..n}\} \equiv_\beta \{l_i' = {E_i'}^{i \in 1..n}\}} \tag{E-RECORD}$$

$$\frac{l_i = l_i' \qquad E_i \equiv_\beta E_i'}{< l_i : {E_i}^{i \in 1..n} > \equiv_\beta < l_i' : {E_i'}^{i \in 1..n} >} \tag{E-VARIANT-SORT}$$

Die Regeln steigen rekursiv in die Unterbäume ab und überprüfen diese auf Äquivalenz. In Verbindung mit der Regel E-RED führt dies dazu, daß die Ausdrücke auf eine vollständige β-Normalform gebracht werden. Mit der Regel E-REFL kann danach die Gleichheit der beiden Normalformen überprüft werden.

Für den Calculus of Constructions reichen die oben aufgeführten Regeln jedoch nicht aus. Betrachten wir die folgenden Ausdrücke:

$$\begin{aligned} T &\stackrel{\text{def}}{=} (x :: Int) \to (\text{if } x \leq 3 \text{ then } Int \text{ else } Bool) \to Int \\ F &\stackrel{\text{def}}{=} \lambda x : \text{Int}.\lambda y : (\text{if } x \leq 3 \text{ then } Int \text{ else } Bool).x \\ e &\stackrel{\text{def}}{=} F \text{ as } T \end{aligned}$$

Die Notation $(x :: Int)$ stammt aus der Programmiersprache *Cayenne* [9]. x wird dadurch gebunden und kann in *Dependent Types* im Rumpf des Typs weiterverwendet werden. $(x :: Int) \to T$ ist also nur syntaktischer Zucker für $\Pi x : Int.T$.
e ist wohlgetypt, wenn der Typ von F zu T äquivalent ist. Die if-Ausdrücke in T und F enthalten beide im Bedingungsteil die abstrakte Variable x. Sie können also nicht reduziert werden, was bedeutet, daß $\equiv_\beta$ zwei if-Ausdrücke auf Äquivalenz testen muß. Für die übrigen Ausdrücke aus LC^x lassen sich leicht analoge Beispiele konstruieren. Aus diesem Grund muß der Äquivalenzbegriff von $\equiv_\beta$ auf allgemeine Ausdrücke erweitert werden. Die zusätzlichen Regeln lauten:

$$\frac{E_1 \equiv_\beta E_1' \qquad E_2 \equiv_\beta E_2'}{E_1 \mathit{op} E_2 \equiv_\beta E_1' \mathit{op} E_2'} \qquad \text{(E-Bop)}$$

$$\frac{E \equiv_\beta E'}{\mathit{op}\, E \equiv_\beta \mathit{op}\, E'} \qquad \text{(E-Uop)}$$

$$\frac{E_1 \equiv_\beta E_1' \qquad E_2 \equiv_\beta E_2' \qquad E_3 \equiv_\beta E_3'}{\text{if } E_1 \text{ then } E_2 \text{ else } E_3 \equiv_\beta \text{if } E_1' \text{ then } E_2' \text{ else } E_3'} \qquad \text{(E-If)}$$

$$\frac{E_1 \equiv_\beta E_1' \qquad E_2 \equiv_\beta E_2'}{\text{[un]fold } E_1\, E_2 \equiv_\beta \text{[un]fold } E_1'\, E_2'} \qquad \text{([Un]Fold)}$$

$$\frac{l = l' \qquad E \equiv_\beta E'}{E.l \equiv_\beta E'.l} \qquad \text{(E-Proj)}$$

$$\frac{l = l' \qquad E_1 \equiv_\beta E_1' \qquad E_2 \equiv_\beta E_2'}{< l = E_1 > \text{ as } E_2 \equiv_\beta < l = E_1' > \text{ as } E_2'} \qquad \text{(E-Variant)}$$

$$\frac{E_1 \equiv_\beta E_1' \qquad E_2 \equiv_\beta E_2'}{E_1 \text{ as } E_2 \equiv_\beta E_1' \text{ as } E_2'} \qquad \text{(E-As)}$$

$$\frac{E \equiv_\beta E' \qquad l_i = l_i' \qquad E_i \equiv_\beta E_i'}{\begin{array}{l}\text{case } E \text{ of } < l_i = x_i > \Rightarrow E_i \,^{i \in 1..n} \\ \equiv_\beta \text{case } E \text{ of } < l_i' = y_i > \Rightarrow E_i' \,^{i \in 1..n}\end{array}} \qquad \text{(E-Case)}$$

4.2 Typregeln für den Basiskalkül

Das Typsystem wird durch eine Menge von Inferenzregeln spezifiziert. Wie üblich bedeutet die Schreibweise $\Gamma \vdash E : A$, daß der Ausdruck E unter der Umgebung Γ den Typ A hat. Die Umgebung bindet freie Bezeichner an Typen, so daß beispielsweise $\{f : Int \to Int, x : Int\} \vdash f\ x : Int$ gilt. Die Regeln für den Basiskalkül sind in Abbildung 2 dargestellt. Sie entsprechen im wesentlichen dem System, welches ursprünglich von Barendregt [1] aufgestellt wurde.

$$\frac{}{\vdash \star : \Box} \quad (\textsc{Star}) \qquad\qquad \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \quad (\textsc{Var})$$

$$\frac{\Gamma \vdash E_1 : T_1 \qquad T_1 \to^{\star}_{\beta} \Pi x : A.B \qquad \Gamma \vdash E_2 : A' \qquad A \equiv_{\beta} A'}{\Gamma \vdash E_1\ E_2 : [x \mapsto E_2]B} \quad (\textsc{App})$$

$$\frac{\Gamma \vdash A : s \qquad \Gamma, x : A \vdash b : B \qquad \Gamma \vdash (\Pi x : A.B)\ : t}{\Gamma \vdash (\lambda x : A.b)\ :\ (\Pi x : A.B)} \quad (\textsc{Abstr})$$

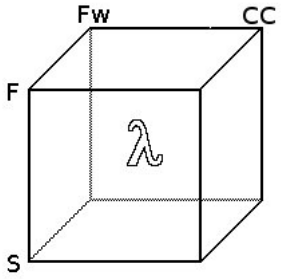
$$\frac{\Gamma \vdash A : s \qquad \Gamma, x : A \vdash B : t \qquad \vdash s \rightsquigarrow t}{\Gamma \vdash (\Pi x : A.B)\ : t} \quad (\textsc{Pi})$$

Abbildung 2: Typregeln für den Basiskalkül

$\star$ ist die Kind-Konstante. Kinds sind die Typen der Sorte Typ. Ihnen wird der Typ $\Box$ zugeordnet. Daß keine Regel für $\Box$ vorhanden ist, impliziert, daß das Symbol nicht als Eingabe im Interpreter vorkommen kann.

Der λ-Cube umfaßt die höherstufigen Kalküle F_ω und CC. Funktionen sind dort auch auf der Ebene von Typen erlaubt – diese werden üblicherweise als Typoperatoren bezeichnet. Daher benötigt man Mechanismen, um die Wohlgetyptheit von Typen zu überprüfen. Diese Aufgabe kommt der Regel Pi zu. Über die $\vdash s \rightsquigarrow t$ Relation kann man „einstellen“, welche mit Π gebildeten Typen man erlauben möchte. Durch Betrachtung von Teilmengen von $\{(\star,\star), (\star,\Box), (\Box,\star), (\Box,\Box)\}$ erhält man unterschiedliche Kalküle. Die Tabelle aus Abbildung 3 listet die für uns relevanten Teilmengen auf. Wenn man davon ausgeht, daß man die Abstraktion auf Termebene und damit $\{(\star,\star)\}$ in jedem Kalkül erlauben möchte, ergeben sich insgesamt acht mögliche Teilmengen. Die acht Teilmengen lassen sich den acht Ecken eines Würfels zuordnen, woraus man eine schöne graphische Darstellung des λ-Cubes erhält.

Mit der Regel Abstr können λ-Abstraktionen typisiert werden. Sie haben immer einen Typ der Form $\Pi x : A.B$. Interessanterweise wird in der Prämisse der Typ auf Wohlgetyptheit überprüft. Über die Regel Pi kann so sichergestellt werden, daß nur Abstraktionen gebildet werden können, die in dem durch $\rightsquigarrow$ festgelegten Kalkül erlaubt sind.



Definition von $\rightsquigarrow$	**Abhängigkeiten**	**Kalkül**
$\{(\star,\star)\}$	Term abhängig von Term	S
$\{(\star,\star),(\Box,\star)\}$	..., Term abhängig von Typ	F
$\{(\star,\star),(\Box,\star),(\Box,\Box)\}$	..., Typ abhängig von Typ	F_ω
$\{(\star,\star),(\Box,\star),(\star,\Box),(\Box,\Box)\}$	..., Typ abhängig von Term	CC

Abbildung 3: λ-Cube und Abhängigkeiten

Im λ-Cube gibt es vier verschiedene Arten von Funktionen. Man kann eine Funktion von Term nach Term, Term nach Typ, Typ nach Term oder Typ nach Typ bilden. Trotzdem genügt App, also genau eine Regel, um die Typkorrektheit der Applikation zu überprüfen. Bei der Applikation von einem Term auf einen Term bewirkt die Substitution auf B nichts, da x nicht frei in B vorkommt. Wird ein Typ auf einen Term angewendet, so wird durch die Substitution die freie Typvariable durch den konkreten Typ ersetzt. Der Typ von E_1 wird auf Normalform reduziert, damit man innerhalb der Regeln auf seine Komponenten zugreifen kann. Der Typ des Formalparameters muß äquivalent zum Typ des Aktualparameters sein.

In Pi und Abstr wird die Typannotation der Variablen zuerst getestet und danach der Kontext erweitert. Dadurch enthält Γ stets nur korrekte Annahmen.

Der einfach getypte λ-Kalkül

Was zeichnet den einfach getypten λ-Kalkül aus und wie wird er in das System des λ-Cubes eingebettet? Wenden wir uns der ersten Frage zu. Die Syntax dieses Kalküls wird normalerweise in der folgenden Form angegeben:

$$\begin{aligned} t &:= x \mid \lambda x : T.t \mid t\,t \\ T &:= X \mid T \to T \end{aligned}$$

Wie bereits erwähnt, läßt sich der Typ $T_1 \to T_2$ schreiben als $\Pi_{\_} : T_1.T_2$. Somit kann die Syntax problemlos in den λ-Cube eingebettet werden. Wir verlieren jedoch die Information, daß die Variable x in der Abstraktion nur mit einem einfachen Typ annotiert werden darf. Die Eigenschaft, daß nur Terme aus Termen abstrahiert werden dürfen, ist nicht mehr durch die Syntax festgelegt. Durch Anpassung der $\rightsquigarrow$ Relation aus Abbildung 3 kann das Typsystem fehlerhafte Abhängigkeiten erkennen. Lassen wir nur $(s,t) \in \{(\star,\star)\}$ zu, so erhalten wir das gewünschte Ergebnis. Mit dieser Einschränkung kann man den Term

$\lambda x : (\Pi X.X \to X).x$ nicht mehr typisieren. Der Typ $\Pi X : \star.X \to X$ würde $\vdash \square \rightsquigarrow \star$ erfordern. Diese Möglichkeit soll in *System F* ausgenutzt werden.

System F

Die typische Syntax von System F lautet:

$$\begin{aligned} t &:= x \mid \lambda x : T.t \mid \lambda X.t \mid t\,t \mid t\,T \\ T &:= X \mid T \to T \mid \forall X.T \end{aligned}$$

$\lambda X.t$ drückt aus, daß es zulässig ist Typen aus Termen zu abstrahieren. Da es in diesem Kalkül nur den trivialen Kind $\star$ gibt, wird üblicherweise auf die Einführung von Kinds verzichtet. Im λ-Cube ist die Annotation jedoch zwingend erforderlich, so daß sich der Typ $\forall X.T$ schreiben läßt als $\Pi X : \star.T$. Um diesen Typ zu überprüfen, benötigen wir $\vdash \square \rightsquigarrow \star$. Die Zusammenlegung der Syntax bedeutet auch, daß wir nur noch eine Form der Applikation benötigen. Die Anzahl der Regeln in der Grammatik reduziert sich somit insgesamt von acht auf vier [1]. Diese Kompaktheit erhöht sich noch in F_ω und CC.

$\mathbf{F}_\omega$

Betrachten wir die übliche Syntax:

$$\begin{aligned} t &:= x \mid \lambda x : T.t \mid \lambda X : K.t \mid t\,t \mid t\,T \\ T &:= X \mid T \to T \mid \forall X : K.T \mid \lambda X : K.T \mid T\,T \\ K &:= \star \mid K \to K \end{aligned}$$

Die Syntax erlaubt die Nutzung von *Typoperatoren*. Diese werden gebildet, indem man Typen aus Typen abstrahiert. Folglich ist $\rightsquigarrow = \{(\star,\star),(\square,\star),(\square,\square)\}$ Gleichzeitig benötigt man eine Applikation auf Typebene zur Eliminierung der Abstraktion. Dadurch erhalten wir auf Typebene die gleichen Konstrukte wie wir sie im einfach getypten λ-Kalkül auf Termebene haben. Um zu verhindern, daß unsinnige Typen gebaut werden können, führt man Kinds ein. Die Anzahl der benötigten Konstrukte wächst dadurch erheblich und es ist offensichtlich, daß viele Formen syntaktisch vollkommen redundant sind (beispielsweise Applikation von Typen auf Terme, von Termen auf Typen und von Typen auf Typen). Im λ-Cube ist keine Trennung der Konzepte auf der Ebene der Syntax vorhanden. Die Syntax ist uniform und dadurch sehr kompakt. Die Einteilung der Ausdrücke in verschiedene Sorten findet erst durch die Typüberprüfung statt.

Calculus of Constructions

Zusätzlich zu F_ω erlaubt man im Calculus of Constructions, Terme aus Typen zu abstrahieren, so daß $\rightsquigarrow$ durch die Menge $\{(\star,\star),(\square,\star),(\star,\square),(\square,\square)\}$ definiert ist. Alle vier Möglichkeiten der Applikation sind somit zulässig. CC wird im Originalartikel [4] mit einer uniformen Syntax eingeführt. Der Basiskalkül des λ-Cubes mit allen Optionen in $\rightsquigarrow$ entspricht genau dem Calculus of Constructions.

[1] Zusätzlich benötigen wir die Konstanten $\star$ und $\square$, die aber zumindest in der externen Syntax vollständig weggelassen werden können.

4.3 Typregeln für einfache Erweiterungen

Der Basiskalkül ist bis auf die Abwesenheit allgemeiner Rekursion schon sehr ausdrucksstark. Allerdings ist es in der Praxis nicht anzustreben, mit Zahlen stets im Church-Encoding zu arbeiten. Dies motiviert die Integration von Basistypen und zugehörigen Werten in den Kalkül. Die Erweiterung erweist sich als vollkommen unproblematisch. Die Unterscheidung zwischen dem Typ *Int* und dem Term *123*, welche aus syntaktischer Sicht beides Ausdrücke sind, stellt für das Typsystem keine Schwierigkeiten dar.
Der λ-Cube soll um boolesche Werte, ganze Zahlen und *unit* erweitert werden. Die neuen Typregeln ergeben sich wiefolgt:

$$\frac{}{\vdash \text{Bool} : \star} \text{ (T-Bool)} \qquad \frac{}{\vdash \text{Int} : \star} \text{ (T-Int)} \qquad \frac{}{\vdash \text{Unit} : \star} \text{ (T-Unit)}$$

$$\frac{}{\vdash \text{true} : \text{Bool}} \text{ (Bool)} \qquad \frac{}{\vdash n : \text{Int}} \text{ (Int)} \qquad \frac{}{\vdash \text{unit} : \text{Unit}} \text{ (Unit)}$$

$$\frac{\Gamma \vdash E : \text{Int}}{\Gamma \vdash \sim E : \text{Int}} \text{ (U-Minus)} \qquad \frac{\Gamma \vdash E : \text{Bool}}{\Gamma \vdash \text{not } E : \text{Bool}} \text{ (Not)}$$

$$\frac{\Gamma \vdash E_1 : \text{Int} \qquad \Gamma \vdash E_2 : \text{Int}}{\Gamma \vdash E_1 \, aop \, E_2 : \text{Int}} \text{ (A-Bin-Op)}$$

$$\frac{\Gamma \vdash E_1 : \text{Int} \qquad \Gamma \vdash E_2 : \text{Int}}{\Gamma \vdash E_1 \, cop \, E_2 : \text{Bool}} \text{ (C-Bin-Op)}$$

Wobei $aop = \{+, -, *\}$ und $cop = \{<>, \leq, \geq, <, >, =\}$ sind.
Die Regeln T-Int, T-Bool und T-Unit zeichnen die neuen Basistypen eindeutig als Typen aus. Den neuen Werten wird der jeweils passende Basistyp zugeordnet. Unäre und binäre Operationen auf den Werten verhalten sich in gewohnter Art und Weise.
Ein Ausdruck kann mit einem Typ annotieren werden, wodurch man eine sogenannte *Zusicherung* erhält. Diese ist nicht auf die Ebene der Terme eingeschränkt. Man kann also auch $(\lambda X.X)$ *as* $(\star \to \star)$ schreiben. Damit hat man die Zusicherung auf Typen ausgedehnt.

$$\frac{\Gamma \vdash E_1 : A \qquad A \equiv_\beta E_2}{\Gamma \vdash E_1 \text{ as } E_2 : E_2} \text{ (Ascribe)}$$

An der Regel für If läßt sich eine interessante Beobachtung machen. Die Syntax von LC^x hat zur Folge, daß in einem If-Konstrukt Terme, Typen und Kinds in Konklusion und Alternative auftreten können. Da die Regel etwas strenger sein soll, reicht die Äquivalenz der Typen von E_1 und E_2 nicht aus.

$$\frac{\begin{array}{c}\Gamma \vdash E_1 : T \qquad T \equiv_\beta \text{Bool} \qquad \Gamma \vdash E_2 : A \\ \Gamma \vdash E_3 : A' \qquad A \equiv_\beta A' \qquad \Gamma \vdash A : s \qquad \vdash \star \rightsquigarrow s\end{array}}{\Gamma \vdash \text{if } E_1 \text{ then } E_2 \text{ else } E_3 : A} \text{ (If)}$$

Nach dem Äquivalenztest muß der Typ von A bestimmt werden. In einem If-Ausdruck hängen die Konsequenz E_2 und die Alternative E_3 von der Bedingung E_1 ab. Um zu überprüfen, ob die Abhängigkeit zulässig ist, prüfen wir, ob $\vdash \star \rightsquigarrow s$ gilt. Der Typ von *Bool* wurde hier direkt als $\star$ angegeben.

4.4 Besonderheiten bei If und Case

Nehmen wir an, wir hätten für alle Kalküle die Regel

$$\frac{\Gamma \vdash E_1 : T \qquad T \equiv_\beta \text{Bool} \qquad \Gamma \vdash E_2 : A \qquad \Gamma \vdash E_3 : A' \qquad A \equiv_\beta A'}{\Gamma \vdash \text{if } E_1 \text{ then } E_2 \text{ else } E_3 : A} \qquad (\text{IF'})$$

eingeführt und betrachten das folgende Beispiel:

$$\lambda b : Bool.\lambda\, x : (\text{if } b \text{ then } Int \text{ else } Bool).x$$

Offensichtlich ist der Typ von x abhängig von dem Wert b, also ein *Dependent Type*. Solche Typen sollten nur im Calculus of Constructions möglich sein. Mit der Regel IF' ist die folgende Ableitung aber in *allen* Kalkülen zulässig:

$$\dfrac{\dfrac{}{\{b : Bool\} \vdash b : Bool}}{\dfrac{\overline{\{b : Bool\} \vdash Int : \star} \qquad \overline{\{b : Bool\} \vdash Bool : \star} \qquad \star \equiv_\beta \star}{\{b : Bool\} \vdash \underbrace{\text{if } b \text{ then } Int \text{ else } Bool}_{=:T} : \star}\ (\text{IF'})}$$

$$\frac{\{T : \star\} \vdash T : \star \qquad \{T : \star, x : T\} \vdash T : \star \qquad \vdash \star \rightsquigarrow \star}{\{T : \star\} \vdash \Pi x : T.T : \star}$$

$$\dfrac{\dfrac{\begin{array}{c}\{b : Bool\} \vdash T : \star \\ \{b : Bool, x : T\} \vdash x : T \\ \{b : Bool\} \vdash (\Pi x : T.T) : \star\end{array}}{\{b : Bool\} \vdash \lambda x : T.x : (\Pi x : T.T)} \qquad \dfrac{\begin{array}{c}\emptyset \vdash Bool : \star \\ \{b : Bool\} \vdash (\Pi x : T.T) : \star \\ \vdash \star \rightsquigarrow \star\end{array}}{\emptyset \vdash \Pi b : Bool.(\Pi x : T.T) : \star}}{\emptyset \vdash \lambda b : Bool.\lambda x : T.x : (\Pi x : Bool.(\Pi y : T.T))}$$

Um diese Ableitung in den Kalkülen S, F und F_ω zu unterbinden, ist die kompliziertere Regel IF notwendig. Für *Case* läßt sich leicht ein ähnliches Beispiel konstruieren.

Betrachten wir

$$\Pi x : Int.\text{if } x \leq 3 \text{ then } Int \text{ else } Bool$$

so stellt sich eine weitere interessante Frage: Wie sieht der Term zu diesem Typ aus? Intuitiv könnten wir den folgenden Term dazu angeben:

$$\lambda x : Int.\text{if } x \leq 3 \text{ then } 12 \text{ else } true$$

Allerdings können wir mit der Regel If hierfür keine Ableitung angeben. Für den Calculus of Constructions können wir eine weitere Regel hinzufügen:

$$\frac{\Gamma \vdash e_1 : T \qquad T \equiv_\beta \mathit{Bool} \qquad \Gamma \vdash e_2 : s_2 \qquad \Gamma \vdash e_3 : s_3}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \text{if } e_1 \text{ then } s_2 \text{ else } s_3} \quad \text{(If-CC)}$$

Damit ist der obige Term typbar. Auf dieser Idee basieren die sogenannten *Proof Assistants* [2]. Da *Cubint* von vornherein aber nicht als Beweis-Tool gedacht war, wurde nicht die allgemeinste Form für die Erweiterungen gewählt. Ziel war es S, F und F_ω möglichst natürlich in den mächtigeren Kalkül einzubetten.

4.5 Regeln für iso-rekursive Typen und Fix

Es wurden iso-rekursive Typen in LC^x integriert. Die Typen $\mu X.X \to T$ und $(\mu X.X \to T) \to T$ werden als ungleich angesehen. Mit den Operationen *fold* und *unfold* kann man den Isomorphismus, der zwischen den beiden Typen besteht, explizit angeben.

$$\frac{\Gamma, X : \star \vdash E : \star}{\Gamma \vdash \mu X.E : \star} \quad \text{(Rec-Type)}$$

$$\frac{\Gamma \vdash E_1 : s \qquad E_1 \to^\star_\beta \mu X.A \qquad \Gamma \vdash E_2 : B \qquad B \equiv_\beta A[X := E_1]}{\Gamma \vdash \text{fold } E_1\, E_2 : E_1} \quad \text{(Fold)}$$

$$\frac{\Gamma \vdash E_1 : s \qquad E_1 \to^\star_\beta \mu X.A \qquad \Gamma \vdash E_2 : B \qquad B \equiv_\beta E_1}{\Gamma \vdash \text{unfold } E_1\, E_2 : B[X := E_1]} \quad \text{(Unfold)}$$

Grundsätzlich ermöglichen rekursive Typen die Definition eines Fixpunktoperators. Ein Fixpunktoperator für rekursive Funktionen über ganze Zahlen kann wiefolgt definiert werden:

$$\begin{aligned} T &\stackrel{\text{def}}{=} \mathit{Int} \to \mathit{Int} \\ \Omega &\stackrel{\text{def}}{=} \lambda x : \mu X.X \to T.\lambda y : \mathit{Int}.f\,((\text{unfold}\,(\mu X.X \to T)\,x)\,x)\,y \\ \text{Fix} &\stackrel{\text{def}}{=} \lambda f : T \to T.\Omega\,(\text{fold}\,(\mu X.X \to T)\,\Omega) \end{aligned}$$

Die Konstruktion ist also recht komplex. Wenn wir im einfach getypten λ-Kalkül sind, hat sie den großen Nachteil, daß man für jeden Funktionstyp einen eigenen Fixpunktoperator definieren muß. Um diesen Overhead zu vermeiden und um unkompliziert Rekursion benutzen zu können, wurde der Fixpunkt-Operator *fix* in das System integriert.

$$\frac{\Gamma \vdash t : T \qquad \Gamma \vdash T : \star \qquad T \to^\star_\beta (\Pi x : T_1.T_2) \qquad T_1 \equiv_\beta T_2}{\Gamma \vdash \mathit{fix}\ t : T_2} \quad \text{(Fix)}$$

Bei den Regeln Fold, Unfold und Fix wird in den Prämissen eine ganz bestimmte Form von Typ gefordert. Daher werden die Typen erst evaluiert.

Mit der Erweiterung um rekursive Typen und einen Fixpunkt-Operator verlieren wir eine zentrale Eigenschaft des λ-Cubes: strenge Normalisierung. Folglich kann die Reduktion wohlgetypter Ausdrücke divergieren. Für den Interpreter bedeutet dies, daß der explizite Aufruf von `equiv E1,E2` für $E_1 \equiv_\beta E_2$ divergieren kann. Abschnitt 7.7 stellt dar, wie man durch geschickte Implementierung zumindest in einigen Fällen die Divergenz vermeiden kann.
Die Regel Rec-Type beschränkt die Rekursion auf Terme und in Fix darf t nur ein Term sein. Type-Checking bleibt dadurch entscheidbar [2] und auf Termebene erhalten wir die Möglichkeit alle Turing-berechenbaren Funktionen auszudrücken. Für spätere Arbeiten wäre es reizvoll zu betrachten, wie sich allgemeine Rekursion auf Typebene auf die Ausdrucksstärke der Kalküle auswirkt.

4.6 Typregeln für Records

Als nächstes sollen Records (deutsch: Verbunde) in den Kalkül integriert werden. Hierfür erweitern wir die Syntax um $\{l_i = E_i{}^{i\in 1..n}\}$ und $\{l_i : E_i{}^{i\in 1..n}\}$. Die Typregeln lauten:

$$\frac{\Gamma \vdash E_i : s_i \qquad i \in 1..n \qquad \Gamma \vdash \{l_i : s_i{}^{i\in 1..n}\} : t}{\Gamma \vdash \{l_i = E_i{}^{i\in 1..n}\} : \{l_i : s_i{}^{i\in 1..n}\}} \qquad \text{(Record)}$$

$$\frac{\Gamma \vdash E_i : s_i \qquad \vdash \star \rightsquigarrow s_i \qquad i \in 1..n}{\Gamma \vdash \{l_i : E_i{}^{i\in 1..n}\} : \star} \qquad \text{(Record-Type)}$$

$$\frac{\Gamma \vdash E : A \qquad A \rightarrow_\beta^\star \{l_i : s_i{}^{i\in 1..n}\} \qquad 1 \leqslant j \leqslant n}{\Gamma \vdash E.l_j : s_j} \qquad \text{(Proj)}$$

Wir legen in der Regel Record-Type fest, daß Records Terme sind. Dadurch gliedern sich Records problemlos in die bestehende Sortenhierarchie ein. Projektion beschreibt den Zugriff auf ein bestimmtes Element des Records. Formal kann man ein Record als eine endliche partielle Funktion $r \in Labels \xrightarrow{\text{fin}} Exp$ ansehen. Das Ergebnis der Projektion entspricht dann dem Funktionswert $r(l_j)$. Record-Type überprüft in der Prämisse, daß aus dem Record-Term nur Ausdrücke extrahiert werden können, die in der $\rightsquigarrow$ Relation zulässig sind. Konkret bedeutet dies, daß in S, F und F_ω nur Records zugelassen sind, die ausschließlich Terme enthalten. In CC gilt $\star \rightsquigarrow \Box$, was bedeutet, daß man Records bauen darf, welche Terme und Typen enthalten. Der Term $\lambda r : \{l : \star\}.\lambda y : (r.l).y$, bei dem der Typ von y von dem Term r abhängt, ist deshalb nur in CC zulässig.

Natürlich sind auch noch andere Formen der Typisierung für Records denkbar. Man kann zum Beispiel festlegen, daß Records ausschließlich Terme oder

[2] Hierbei handelt es sich nur um eine starke Vermutung. Ob Type-Checking in LC^x wirklich entscheidbar ist, kann natürlich nur in einem formalen Beweis geklärt werden, der aber den Rahmen dieser Arbeit sprengt.

ausschließlich Typen enthalten können. Ein Record von Termen kann dann eindeutig als Term klassifiziert werden und ein Record von Typen als Typ. Die Regel RECORD-TYPE wird durch zwei Regeln ersetzt.

$$\frac{\Gamma \vdash E_i : \star \qquad i \in 1..n}{\Gamma \vdash \{l_i : {E_i}^{\,i \in 1..n}\} : \star} \qquad \frac{\Gamma \vdash E_i : \Box \qquad i \in 1..n}{\Gamma \vdash \{l_i : {E_i}^{\,i \in 1..n}\} : \Box}$$

Mit Typ-Records kann man in F_ω kaskadierte Typoperatoren durch kartesische ersetzten. Der Typoperator

$$\lambda X : \star . \lambda Y : \star . \, X \to Y$$

kann dann wiefolgt geschrieben werden:

$$\lambda R : \{l_1 : \star, l_2 : \star\} . \, R.l_1 \to R.l_2$$

Cubint verwendet nicht die alternativen Regeln, sondern das zuerst eingeführte System. Dadurch besitzt man in S, F und F_ω genau die Möglichkeiten, welche durch Integration von Records in die klassische Syntax entstehen. Zusätzlich hat man den Vorteil, daß die Regeln syntaxgerichtet sind.

4.7 Typregeln für Varianten

Beim Programmieren ist es oft nützlich mit einer Ansammlung inhomogener Werte zu arbeiten. Insbesondere im Zusammenhang mit rekursiven Typen lassen sich dadurch Datenstrukturen wie Listen oder Bäume definieren. Der Typ für eine Liste kann zum Beispiel wiefolgt aussehen:

$$\text{List} \stackrel{\text{def}}{=} \mu X. < \text{nil} : \text{Unit}, \text{cons} : \{\text{hd} : \text{Int}, \text{tl} : X\} >$$

Werte dieses Typs sind entweder *Unit* oder ein Record-Typ. Über die Labels *nil* und *cons* kann man in einem *case*-Ausdruck überprüfen, welcher Wert in der Variante tatsächlich gespeichert ist. Die Typregeln für Varianten sind vom Aufbau denen von Records ähnlich.

$$\frac{\begin{array}{c}\Gamma \vdash E_1 : A \qquad \Gamma \vdash E_2 : s \qquad E_2 \to^\star_\beta < l_i : {B_i}^{\,i \in 1..n} > \\ l = l_j \qquad A \equiv_\beta B_j \qquad j \in 1..n\end{array}}{\Gamma \vdash < l = E_1 > \text{ as } E_2 : E_2} \qquad \text{(VARIANT)}$$

$$\frac{\Gamma \vdash E_i : s_i \qquad \vdash \star \rightsquigarrow s_i \qquad i \in 1..n}{\Gamma \vdash < l_i : {E_i}^{\,i \in 1..n} > : \star} \qquad \text{(VARIANT-TYPE)}$$

$$\frac{\begin{array}{c}\Gamma \vdash E : A \qquad A \to^\star_\beta < l_i : {s_i}^{\,i \in 1..n} > \\ \Gamma, x_i : s_i \vdash E_i : A_i \qquad A_i \equiv_\beta A_{i+1} \\ \Gamma \vdash A_i : s_i \qquad \vdash \star \rightsquigarrow s_i\end{array}}{\Gamma \vdash \text{case } E \text{ of } < l_i = x_i > \Rightarrow {E_i}^{\,i \in 1..n} : A_i} \qquad \text{(CASE)}$$

Die Zusicherung an der Variante wird unbedingt benötigt. Nur so kann man herausfinden, welche anderen Werte von Ausdrücken dieses Typs angenommen werden können. Die Zusicherung ermöglicht es, dem Ausdruck einen eindeutigen Typ zuzuweisen. Erst wenn Subtyping integriert wird, kann man sie weglassen (vergleiche *Types and Programming Languages* [11], Seite 196-197). Die Regel VARIANT-TYPE erlaubt, daß in CC Varianten zulässig sind, die Terme und Typen enthalten.
In der Regel CASE müssen wir, wie aus Abschnitt 4.4 bekannt ist, überprüfen, ob $\star \rightsquigarrow s_i$ gilt. Da Varianten als Terme ausgezeichnet werden, steht $\star$ für die Variante, für die die Fallunterscheidung vorgenommen wird.

4.8 Typregeln für Referenzen

Bisher haben wir nur rein funktionale Elemente von Programmiersprachen betrachtet. Imperative Programmiersprachen beinhalten Objekte, die auf einen Wert verweisen. Die Verweise können im Laufe des Programms verändert werden, so daß das Objekt verschiedene *Zustände* annehmen kann. Dieser Abschnitt beschreibt die Integration von Referenzen in LC^x. Unter einer Referenz verstehen wir eine abstrakte Speicherzelle. Die Menge Σ aller Speicherzellen bezeichnen wir als *Speicher*. Wir erweitern nun die Syntax der Ausdrücke und Werte:

$$
\begin{array}{lcll}
E & ::= & \ldots & \\
 & | & \text{Ref } E & \text{Referenz-Typ} \\
 & | & \text{ref } E & \text{Allokation} \\
 & | & !\,E & \text{Dereferenzierung} \\
 & | & E := E & \text{Zuweisung} \\
 & | & loc & \text{Speicherzelle} \\
 & & & \\
V & ::= & \ldots & \\
 & | & loc & \text{Speicherzelle}
\end{array}
$$

Die Basisoperationen auf Referenzen sind Allokation, Dereferenzierung und Zuweisung. Mit *ref E* kann eine neue Speicherzelle alloziert werden, die den Wert von E enthält. Als Resultat der Auswertung von *ref E* erhält man die Adresse *loc*, unter der die Zelle in Σ erreichbar ist. Per Dereferenzierung kann der Zustand einer Speicherzelle abgefragt und mit dem Zuweisungsoperator $:=$ verändert werden. Die Berechnung des Ausdrucks $x := E$ liefert keinen Wert, sondern verändert den Zustand der Speicherzelle, die an x gebunden ist. Man spricht davon, daß die Berechnung einen *Seiteneffekt* hat. Bei der Auswertung der Ausdrücke kann der Zustand des Speichers verändert werden. Für die Zuweisung erhalten wir unter anderem die Regel

$$loc := v \,|\, \Sigma \rightarrow \text{unit} \,|\, [loc \mapsto v]\Sigma$$

Im Kontext von Referenzen erhält der Wert *unit* seinen ersten sinnvollen Einsatz. Die weiteren Evaluationsregeln können in *Types and Programming Languages* [11] S.161 - 162 nachgelesen werden.
Mithilfe von Referenzen kann ein Fixpunktoperator definiert werden. Um mit den Regeln für iso-rekursive Typen und fix konform zu gehen, erlauben wir daher nur Referenzen auf der Ebene von Termen. Wir erhalten die folgenden Typregeln:

$$\frac{\Gamma \vdash E : \star}{\Gamma \vdash \mathrm{Ref}\, E : \star} \qquad \text{(Ref-Type)}$$

$$\frac{\Gamma \vdash E : T \qquad \Gamma \vdash \mathrm{Ref}\, T : s}{\Gamma \vdash \mathrm{ref}\, E : \mathrm{Ref}\, T} \qquad \text{(Ref)}$$

$$\frac{\Gamma \vdash E : \mathrm{Ref}\, T}{\Gamma \vdash !E : T} \qquad \text{(Deref)}$$

$$\frac{\Gamma \vdash E_1 : \mathrm{Ref}\, T_1 \qquad \Gamma \vdash E_2 : T_2 \qquad T_1 \equiv_\beta T_2}{\Gamma \vdash E_1 := E_2 : \mathrm{Unit}} \qquad \text{(Assign)}$$

Durch Anpassung der Regel Ref-Type könnten wir Referenzen auf Typebene zulassen. Hier müßten aber zuerst die Auswirkungen solch einer Änderung untersucht werden.

5 Typrekonstruktion

Der Nutzen statischer Typsysteme für Programmiersprachen ist unbestritten. Mit einem Typsystem können während der Compile-Zeit erstaunlich viele Programmierfehler aufgedeckt werden. Wenn der Programmierer jeden Typ stets vollständig angeben muß, kann allerdings ein erheblicher Mehraufwand entstehen. Daher wurden Typsysteme, wie beispielsweise das von ML, entworfen, die es weitgehend ermöglichen den ungetypten λ-Kalkül einzugeben. Die fehlenden Typangaben werden von einem Inferenzalgorithmus ergänzt. Die Identitätsfunktion kann daher wiefolgt geschrieben werden:

$$id \stackrel{\text{def}}{=} \lambda x.x$$

Hierbei stellt sich die Frage, welcher Typ für *id* abgeleitet werden soll. Ein spezieller Typ $Int \to Int$ ist nicht wünschenswert. Es wäre nützlich, wenn man die Funktion *polymorph*, also mit verschiedenen Typen, verwenden könnte. Polymorphismus kennen wir schon aus System F. Die Frage, ob es zu einem ungetypten Ausdruck einen Typ aus System F gibt, ist aber unentscheidbar. In diesem Abschnitt soll ein System beschrieben werden, welches dem von ML ähnlich ist und von der Mächtigkeit her zwischen dem einfach getypten λ-Kalkül und System F liegt. Damit wird die Typrekonstruktion entscheidbar und man erhält eine etwas schwächere Form von Polymorphismus.
Zuerst wird die Typrekonstruktion in *Cubint* mit der von ML verglichen und Gemeinsamkeiten und Unterschiede herausgearbeitet. Dann wird ein Algorithmus vorgestellt mit dem zu einem Ausdruck der allgemeinste Typ bestimmt werden kann.

5.1 Vergleich mit ML

Der Typinferenz-Algorithmus in ML ist über Typschemata realisiert. Der klassische Algorithmus hierfür ist *Algorithm W* von Damas und Milner [5]. Er leitet

für *Let*-gebundene Ausdrücke einen allgemeinsten Typ her. Dadurch entsteht eine einfache Form des Polymorphismus, die man *Let-Polymorphismus* nennt. Schauen wir uns zur Verdeutlichung ein Beispiel an:

$$\begin{array}{lll} \text{let} & id = \lambda x.x & \text{in} \\ \text{let} & x = id\ \ 3 & \text{in} \quad id\ \ true \end{array}$$

Damit *id* polymorph verwendet werden kann, weist der Algorithmus der Funktion das Typschema $\forall\alpha.\alpha \to \alpha$ zu. Dieses wird an der passenden Stelle mit *Int* und *Bool* instantiiert. Typvariablen treten also nur in gebundener Form innerhalb von Typschemata auf.
Cubint wählt eine andere Variante, erreicht aber das gleiche Ziel. Grundlage der Typüberprüfung ist die Regel T-Letpoly.

$$\frac{\Gamma \vdash [x \mapsto e_1]e_2 : T_2 \qquad \Gamma \vdash e_1 : T_1}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : T_2} \quad \text{(T-Letpoly)}$$

Im Rumpf e_2 des let-Ausdrucks wird jedes Vorkommen der Variable x durch ihre Definition e_1 ersetzt. Anschließend bestimmt man den Typ des Ausdrucks $[x \mapsto e_1]e_2$. Der Typ von e_1 muß separat überprüft werden. Wenn x nicht in e_2 vorkommt, würde sonst ein eventuell inkorrektes Programm akzeptiert werden. Typschemata kommen bei diesem Vorgehen nicht zum Einsatz. Die Definition e_1 wird nämlich im Prinzip gar nicht polymorph verwendet. Vielmehr wird in jedem Kontext ein neuer Typ für den Ausdruck hergeleitet. Eine Bindung von Typvariablen ist nicht notwendig. Wir können also $id : \alpha \to \alpha$ herleiten.

5.2 Constraint-basierte Typisierung

Die Frage, wie der allgemeinste Typ für *id* hergeleitet wird, haben wir bisher noch nicht beantwortet. Der Algorithmus in *Cubint* inferiert Constraints, die durch Unifikation gelöst werden. Eine detaillierte Beschreibung dieses Systems ist in *Types and Programming Languages* [11], S. 322-329, zu findet. Der Typ eines Ausdrucks wird in drei Schritten bestimmt:

1. Der Inferenzalgorithmus erzeugt eine Menge von Constraints und einen Typ
$$\Gamma \vdash e : T \mid \mathcal{C}$$

2. Zur Lösung der Contraints $\mathcal{C}$ wird Unifikation verwendet. Die Lösung entspricht einer Substitution der Form
$$\sigma = \{X_1 = T_1, \ldots, X_n = T_n\}$$

3. Der allgemeinste Typ für e entspricht
$$\sigma(T)$$

Schauen wir uns exemplarisch zwei Regeln an, um den ersten Schritt nachzuvollziehen.

$$\frac{\Gamma, x : T_1 \vdash e : T_2 \mid \mathcal{C}}{\Gamma \vdash \lambda x : T_1.e : T_1 \to T_2 \mid \mathcal{C}} \qquad \text{(Ct-Abs)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : T_1 \mid \mathcal{C}_1 \qquad \Gamma \vdash e_2 : T_2 \mid \mathcal{C}_2 \\ \mathcal{C}' = \mathcal{C}_2 \cup \mathcal{C}_2 \cup \{T_1 = T_2 \to X\} \qquad X \text{ fresh}\end{array}}{\Gamma \vdash e_1\ e_2 : X \mid \mathcal{C}'} \qquad \text{(Ct-App)}$$

Der Algorithmus erkennt keine falsch getypten Terme, sondern vermerkt, welche Bedingungen für die Wohlgetyptheit des Ausdrucks erfüllt sein müssen. Für die Applikation muß gelten, daß e_1 eine Funktion ist, auf die ein Argument vom Typ T_2 appliziert werden darf. Diese Bedingung wird durch $\{T_1 = T_2 \to X\}$ protokolliert, wobei X eine neue (*fresh*) Typvariable ist[3]. Die restlichen Regeln sind nach dem gleichen Prinzip aufgebaut. Das Verfahren liefert für jeden Ausdruck eine Menge von Bedingungen. Unifikation überprüfen, ob die Constraint-Menge Widersprüche enthält, und erzeugt dabei eine Substitution σ. $\sigma(X)$ ist der allgemeinste Typ der Applikation $e_1\ e_2$.

In *Cubint* ist der Algorithmus auf zwei verschiedene Arten umgesetzt worden. In der ersten Variante arbeitet er direkt auf den Syntaxbäumen der Ausdrücke. Er ist vollkommen analog zum obigen Verfahren aufgebaut. Die zweite Variante basiert auf einer Graphdarstellung von Typen. Die drei Phasen können zu einem Durchlauf verschmolzen werden, was den Algorithmus sehr effizient macht. Genaueres zur Implementierung findet man in Abschnitt 7.6.

5.3 Unifikation und equi-rekursive Typen

Interessanterweise können durch die Graphdarstellung von Typen ohne großen Mehraufwand equi-rekursive Typen in das Typsystem integriert werden. Im Standardalgorithmus für Unifikation [10] werden keine zyklischen Constraints zugelassen. Stößt man auf eine Gleichung der Form $X = T$, wobei $X \in FV(T)$, so bricht der Algorithmus ab. Rekursive Typen beschreiben zyklische Strukturen, weshalb mit ihnen zyklische Constraints sinnvoll behandelt werden können. Man läßt den Zyklencheck im Unfikationsalgorithmus einfach weg und baut eine zusätzliche Überprüfung ein, um die Terminierung sicherzustellen.

6 Subtyping

Sind in einer Programmiersprache Records und Varianten verfügbar, so stellen sich einige interessante Fragen. Welche Argumente können beispielsweise auf die Funktion $f := \lambda r : \{a : Int\}.(r.a + 1)$ angewendet werden? Aus unserer bisherigen Sicht kommt dafür nur ein Record der Form $\{a = number\}$ in Frage. Die Typannotation legt also immer genau fest, wie das Argument auszusehen hat. Diese Sicht wird in den meisten ML Systemen verfolgt. Doch können wir die Anforderung an das Argument deutlich relaxieren ohne die Typsicherheit unseres Systems zu zerstören. Die Funktion f benötigt ein Record, welches das

[3]Die Eigenschaft *fresh* kann formal präzisiert werden, was aber die Regeln deutlich verkompliziert.

Feld a enthält. Wenn dies sichergestellt ist, spielt es keine Rolle, welche Felder zusätzlich noch in dem Record vorkommen. Allgemeiner kann man mit dieser Sichtweise eine partielle Ordnung auf den Typen definieren. Für unser Beispiel bedeutet dies, daß wir alle Record-Typen, die mindestens das Feld a enthalten als kleiner beziehungsweise weniger allgemein als den Typ $\{a : Int\}$ ansehen.
Varianten verhalten sich genau gegensätzlich zu Records. Sei eine Funktion

$$g := \lambda v :< a : Int, b : Bool > .case\ v\ of < a = x > \Rightarrow 1 \mid < b = x > \Rightarrow 2$$

gegeben. Dann kann g alle Argumente aufnehmen, deren Typ eine Teilmenge von $< a : Int, b : Bool >$ darstellt. Je mehr Felder also ein Varianten-Typ hat, desto allgemeiner ist er.
Subtyping auf Records und Varianten, wie es oben beschrieben wurde, nennt man *strukturell*, im Gegensatz zu *nominellen* Systemen, die in objektorientierten Sprachen häufig zum Einsatz kommen. In nominellen Systemen wird die Hierarchie explizit über Vererbung festgelegt.
Bevor wir zu den Details kommen, soll ein wesentlicher Nachteil von Subtyping angesprochen werden. Subtyping ist häufig mit Informationsverlust verbunden. Schleust man das Record $\{a = 3, b = 4\}$ durch die Funktion $\lambda r : \{a : Int\}.r$, so kommt jegliche Typinformation über das Feld b abhanden. Um diese ungünstige Eigenschaft auszubügeln, gibt es in vielen Sprachen einen sogenannten *Downcast*. Möchte man in solch einem System Typsicherheit erhalten, ist man auf dynamische Tests angewiesen[4]. Da in *Cubint* die Typüberprüfung statisch sein soll, wurde auf Downcasts verzichtet.

6.1 Subtyping in *Cubint*

Die Integration von Subtyping sollte ohne große Änderungen an der Struktur des Interpreters vorsichgehen. Glücklicherweise erwies sich diese Anforderung als unproblematisch. Die Typregeln können auf Subtyping angepaßt werden, indem man die Relation $\equiv_\beta$ durch die Subtyping-Relation $<:$ ersetzt. Da die neue Relation nicht mehr symmetrisch ist, muß man darauf achten, die Ausdrücke in der richtigen Reihenfolge zu prüfen. Die Regel für die Applikation ergibt sich dann wiefolgt:

$$\frac{\Gamma \vdash E_1 : A \qquad A \rightarrow_\beta^\star \Pi x : A.B \qquad \Gamma \vdash E_2 : A' \qquad A' <: A}{\Gamma \vdash E_1\, E_2 : B[x := E_2]} \quad (\textsc{App})$$

Für höherstufigen Typsysteme ist es interessant weitere Features wie *bounded quantification* zu betrachten. In solchen Systemen kann zusätzlich die folgende Art von Typen definiert werden: $\forall X <: T_1.T_2$. Dieser Typ macht ein neues Konstrukt erforderlich, welches nur bei eingeschaltetem Subtyping Sinn macht. Um dies zu vermeiden, wurde auf das Feature verzichtet.
Wie wir in Abschnitt 4.1 gesehen haben, mußten wir für den Calculus of Constructions $\equiv_\beta$ auf alle Ausdrücke erweitern. Wie aber soll $<:$ auf allgemeine Ausdrücke erweitert werden? Kann man auch zwei If-Ausdrücke oder zwei Zahlen mit $<:$ vergleichen? Die Subtyping Relation kann offensichtlich nicht analog

[4]Diese Tests sind ein Grund dafür, weshalb die Effizienz von Java-Programmen oft nicht mit der von C++ Programmen mithalten kann.

zu $\equiv_\beta$ verallgemeinert werden. Wie der Name schon sagt, bezieht sich die Relation prinzipiell nur auf Typen. Eine Erweiterung auf Ausdrücke der Sorte Term scheint ein großer Schritt zu sein und soll deshalb im Rahmen dieser Arbeit nicht behandelt werden.

6.2 Subtyping auf iso-rekursiven Typen

Um Subtyping mit iso-rekursiven Typen zu verknüpfen kann die Relation $<:$ durch die von Cardelli entwickelte *Amber rule* [3] ergänzt werden.

$$\frac{\Sigma, X <: Y \vdash S <: T}{\Sigma \vdash \mu X.S <: \mu Y.T} \qquad \text{(S-Amber)}$$

$$\frac{(X <: Y) \in \Sigma}{\Sigma \vdash X <: Y} \qquad \text{(S-Assumption)}$$

$\Sigma \vdash \mu X.S <: \mu Y.T$ gilt, wenn $\Sigma, X <: Y \vdash S <: T$. Diese Regeln setzen voraus, daß X und Y verschiedene Variablen sind, damit die linke und die rechte Seite der Relation klar unterschieden werden können. Ansonsten könnte man nicht feststellen, ob $X <: Y$ oder $Y <: X$ gilt. S-Amber besagt, daß $\mu X.S <: \mu Y.T$ gilt, wenn $S <: T$ unter der Annahme $X <: Y$ gilt.

> **Besonderheit in de Bruijn-Notation**
> Die Tatsache, daß X und Y verschieden sein müssen, kollidiert mit der de Bruijn-Darstellung von Variablen. Die beiden gebundenen Namen X und Y werden nämlich nicht unterschieden. Zur Lösung verwaltet man zwei Kontexte. In den Regeln wird Σ durch $\Sigma_1 \mid \Sigma_2$ ersetzt und die neue Annahme in S-Amber in Σ_1 eingefügt. Bei S-Assumption wird die Annahme nur im ersten Kontext gesucht. Da die Position der Annahme im Kontext die Nummer der Variable repräsentiert, speichert man anstelle von $X <: T$ nur den Typ T. In einer kontravarianten Regel werden die Kontexte dann einfach ausgetauscht.
>
> $$\frac{\Sigma_2|\Sigma_1 \vdash T_1' <: T_1 \qquad \Sigma_1, T_1'|\Sigma_2, T_1' \vdash T_2 <: T_2'}{\Sigma_1|\Sigma_2 \vdash \Pi x : T_1.T_2 <: \Pi y : T_1'.T_2'} \qquad \text{(S-Pi)}$$
>
> Man kodiert also die Richtung der $<:$ Relation durch zwei voneinander getrennte Kontexte.

6.3 Supremum und Infimum

Die Supremumsfunktion stellt fest, ob es zu zwei Typen einen gemeinsamen Supertyp gibt. Sie ist verschränkt rekursiv mit einer Infimumsfunktion definiert, um das Supremum von Funktionstypen bestimmen zu können. Die Supremumsfunktion wird bei If und Case eingesetzt. Bei If wird nicht mehr getestet, ob die Typen von Konklusion und Alternative äquivalent sind, sondern ob es einen gemeinsamen Supertyp gibt. Da $<:$ nicht als totale Ordnung definiert wurde,

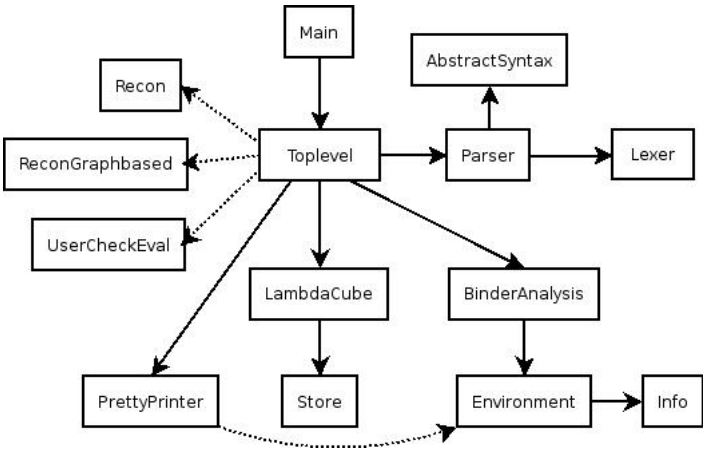


Abbildung 4: Strukturübersicht

existiert möglicherweise kein solcher Typ. In manchen Sprachen gibt es einen Typ *Top*, der die Relation nach oben hin abschließt. Für *Cubint* sprechen zwei Dinge gegen die Integration dieses Typs. Als erstes würde man dadurch ein Subtyping-spezifisches Konstrukt integrieren. Als zweites gibt es in *Cubint* keinen Downcast. Mit einem Ausdruck vom Typ *Top* kann also nicht mehr weitergerechnet werden. Er weist also gewisse Ähnlichkeiten zu einem *stuck term* auf.

7 Die Architektur von *Cubint*

Beim Entwurf des Interpreters lag der Schwerpunkt auf *Einfachheit* und *Modularität*. Das Programm soll Studenten dabei helfen getypte λ-Kalküle, und damit Grundlagen von Programmiersprachen, besser zu verstehen. Ebenso soll es nach kurzer Einarbeitungszeit möglich sein, einen eigenen Type-Checker für *Cubint* zu implementieren. Effizienz stand nicht im Vordergrund, da *Cubint* nicht den Anspruch eines Programmiersystems erfüllen soll. Das Design orientiert sich wesentlich an theoretischen Modellen, so daß beispielsweise β-Reduktion mit Substitution der effizienteren Variante mit Umgebungen vorgezogen wurde.

Anhand von Abbildung 4 soll die Struktur des Entwurfs erläutert werden.

7.1 Einmal durch den Interpreter und zurück

Der Interpreter startet im Modul *Main*. Es enthält die Prozedur *evalCLArgs*, mit der die Kommandozeilenargumente ausgewertet werden können. Außerdem lässt diese Prozedur eventuelle Eingabedateien abarbeiten und liefert eine entsprechend modifizierte Umgebung zurück. Anschließend wird die Prozedur *run* aufgerufen, um in den interaktiven Modus von *Cubint* zu wechseln. *run* stellt die Hauptschleife des Programms dar. Ein Schleifendurchlauf enthält dabei die folgenden Schritte:

1. Der Parser wird auf die Standardeingabe `<stdin>` angesetzt.
2. Der Parser liefert in Kombination mit dem Lexer eine Liste auszuführender Kommandos zurück.
3. Die Kommandos werden nacheinander mit der Prozedur *executeCommand* abgearbeitet und in jedem Schritt eine angepasste Umgebung zurückgegeben.
4. Ausnahmen, die dabei auftreten können, werden von einem *errorhandler* aufgefangen. Dieser gibt eine Fehlermeldung aus und springt danach in den nächsten Schleifendurchlauf.

Es gibt zwei Arten von Kommandos, die von der Prozedur *executeCommand* unterschiedlich behandelt werden. Die erste Art modifiziert die Einstellungen des Interpreters. Hierunter fallen das Umstellen des Typmodus, die Einstellungen zur Ausgabe und das Ein- und Ausschalten von Subtyping.
Ausdrücke aus den verschiedenen Kalkülen stellen die zweite Art von Kommandos dar. Zuerst werden alle freien Bezeichner mit der Prozedur *resolveFreeBinder* anhand der Umgebung aufgelöst. Dabei handelt es sich um eine reine Textersetzung, das heißt, der Bezeichner wird durch seine Definition ersetzt. So erhält man nach dieser Phase stets geschlossene Ausdrücke. Für ein praktisches Programmiersystem ist ein solches Vorgehen natürlich nicht akzeptabel, da man dadurch im schlimmsten Fall exponentiell große Ausdrücke erhält. Für *Cubint* erweist sich diese Entscheidung aus mehreren Gründen als nützlich.

1. Der Kern des Interpreters kommt vollständig ohne globale Umgebung aus. Dies erlaubt eine kompakte Formulierung der Evaluationsfunktion sowie des Type-Checkers.
2. Die Implementierung eines Type-Checkers und Evaluators durch einen Benutzer wird einfacher, da man nur auf geschlossenen Ausrücken arbeitet.
3. Typrekonstruktion läßt sich leicht in das System integrieren. Das Verfahren aus Abschnitt 5.2 kann eins zu eins umgesetzt werden. Da der Parser an jeder Stelle freie Variablen durch ihre Definition ersetzt, muß die Regel T-Letpoly nicht explizit eingeführt werden.

Der geschlossene Ausdruck wird auf Typkorrektheit überprüft [5]. Danach wird der Ausdruck ausgewertet, falls es das Kommando erforderlich macht. Das Kommando `rdef x=3+3` führt beispielsweise dazu, daß die Addition ausgewertet und das Ergebnis an x gebunden wird. `def x=3+3` bindet hingegen x an den Ausdruck $3 + 3$. Der Ausdruck wird zusammen mit seinem Typ durch den Pretty-Printer ausgegeben. Anschließend beginnt der Zyklus wieder von vorn.

7.2 Lexer und Parser

Der Interpreter enthält für alle Kalküle einen gemeinsamen Parser. Die externe Syntax basiert auf der uniformen Syntax des λ-Cubes. Darüber hinaus werden einige Abkürzungen angeboten, die eine komfortablere Eingabe von Ausdrücken erlauben. Eine genaue Beschreibung und Beispiele können im *Userguide* gefunden werden. Der Parser ist auf das mächtigste Kalkül, den Calculus of Constructions, zugeschnitten. Dies hat zur Konsequenz, daß er keinen

[5] Ausnahme ist hierbei der *untyped* Modus der Interpreters.

Fehler meldet, wenn beispielsweise im einfach getypten λ-Kalkül der Ausdruck $\lambda T : \star \rightarrow \star . \lambda x : T\ Int.x$ definiert wird. Erst der Type-Checker meldet einen Fehler. Diese späte Fehlererkennung offenbart, daß die Eingabe einen grundlegenden semantischen Fehler enthält und nicht nur eine syntaktische Ungereimtheit.
Der Parser überführt die Eingaben des Benutzers in die interne Syntax aus Abschnitt 2.

Cubint wurde in der Programmiersprache MoscowML implementiert. Die MoscowML Umgebung stellt Tools zur automatischen Generierung von Lexer und Parser zur Verfügung. *mosmllex* wurde zur Generierung des Lexers verwendet, *mosmlyac* zur Erzeugung des Parsers.

7.3 PrettyPrinter

Der vom Parser generierte Syntaxbaum eignet sich nicht als Ausgabe des Interpreters. Der PrettyPrinter transformiert den Baum in eine Zeichendarstellung aus der Eingabesprache. Dabei wird auf eine minimale Klammerung der Ausdrücke geachtet. Anstelle von `(((3*3)*(3-1))+4)` wird daher `3*3*(3-1)+4` ausgegeben.
Zusätzlich kann der *Cubint*-PrettyPrinter β-Redexe markieren:

```
- def inc = \x:Int.x+1;
> def inc = <fun> : Int->Int
- protocol (inc (2+1));
> def it = inc (2+1) : Int
                ^^^
> def it = inc 3 : Int
           ^^^^^
> def it = 3+1 : Int
           ^^^
> def it = 4 : Int
-
```

7.4 Die Umgebung

Ausdrücke können an Bezeichner gebunden und dadurch in einer Umgebung gespeichert werden. Sie besteht aus einer *Binarymap* mit der man einfach per Bezeichnernamen auf den zugehörigen Ausdruck zugreifen kann. Der Parser markiert ungebundene Variablen, so daß die Prozedur *resolveFreeBinder* aus dem Modul *BinderAnalysis* feststellen kann, welche Variable ersetzt werden soll. Natürlich möchte man, daß bei der Ausgabe die Bezeichner erhalten bleiben und nicht zwangsläufig die Definition offengelegt wird. Deshalb ist jeder Ausdruck mit einem zusätzlichen *info* Feld ausgestattet, welches Informationen über den Bezeichner, an den der Ausdruck gebunden ist, enthält. Das Feld wird gesetzt, sobald ein Ausdruck in die Umgebung eingefügt wird.
Die Umgebung bietet die Möglichkeit den zu einem Ausdruck gehörigen Bezeichner zu finden. Startet man die Suche mit einem Ausdruck E, so wird überprüft, ob die Umgebung einen Ausdruck E' enthält, so daß $E \equiv_\beta E'$ gilt. Diese Art der Suche ist für den PrettyPrinter interessant. Wird der PrettyPrinter auf einen

entsprechenden Modus umgeschaltet, so kann er Typdefinitionen aufspüren.

```
- switch pm ows;
! PP: mode changed to opaque, with environment sync
- def MyType = Int->Int
> def MyType = Int->Int : *
- def f = \x:Int.x+1;
> def f = it : MyType
-
```

Die Äquivalenz von Ausdrücken ist wegen des integrierten Fixpunkt-Operators und der rekursiven Typen unentscheidbar. Deshalb kann es in diesem Ausgabemodus häufig dazu kommen, daß der PrettyPrinter divergiert.

7.5 Syntax-basierte Typrekonstruktion

Das syntax-basierte Verfahren arbeitet direkt auf der Baumdarstellung, den der Parser erzeugt. Dieser Modus erlaubt die Eingabe freier Typvariablen.
Der Algorithmus realisiert ein zum Let-Polymorphismus aus ML ähnliches System. In *Cubint* können neue Bindungen über das Kommando `def x = exp` eingeführt werden. Eine Sequenz dieser Kommandos kann man als geschachtelten Let-Ausdruck ansehen. Wie bereits erwähnt, ist es nicht nötig die Regel T-LETPOLY explizit einzuführen.
Der Type-Checker und die Verwaltung der freien Typvariablen sind im Modul *Recon* zusammengefaßt.

7.6 Graph-basierte Typrekonstruktion

In der Praxis erweist sich das syntax-basierte Verfahren als sehr ineffizient. Schuld daran ist im wesentlichen die Unifikation. Stößt der Algorithmus auf eine Gleichung der Form $X = T$, so muß in der gesamten Menge der Constraints die Variable x durch den Typ T ersetzt werden. Außerdem werden aus einem Constraint der Form $T_1 \to T_2 = S_1 \to S_2$ zwei Constraints $T_1 = S_1$ und $T_2 = S_2$. Im schlimmsten Fall erhält man dadurch exponentielle Laufzeit. Deutlich besser sieht es aus, wenn man Typen nicht als Bäume, sondern als Graphen darstellt. Der erste Vorteil ist, daß man Typen dadurch wesentlich kompakter speichern kann, da äquivalente Unterbäume jeweils nur einmal im Graph auftreten (vergleiche Abbildung 5). Der zweite Vorteil ist, daß die Unifikation direkt auf dem Graphen arbeiten kann. Dadurch muß keine Substitution mehr erzeugt werden und nach der Unifikation kann der fertige Typ direkt ausgeben werden. Die drei Phasen aus Abschnitt 5.2 können also zu einem einzelnen Durchlauf vereinigt werden und die Laufzeit wird linear in der Anzahl der Knoten des Graphs.

Graphen werden als (dynamische) Arrays repräsentiert. Die Knoten des Graphs entsprechen den Nummern der Zellen. In diesen ist gespeichert, ob es sich um einen Knoten mit keinem, einem oder zwei Nachfolgern handelt. Abbildung 5 zeigt, wie man sich die neue Repräsentation vorstellen kann.

Der Typalgorithmus gibt einen Graphen[6] aus, während die übrigen Komponenten von *Cubint* auf einem Baum arbeiten. Auch der Parser von *Cubint* liefert

[6] Ab jetzt meinen wir mit *Graph* seine Repräsentation als Array.

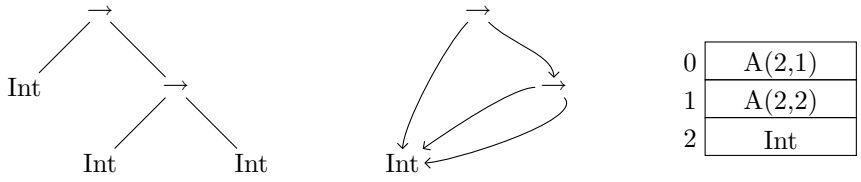


Abbildung 5: Baum-, Graph- und Array-Darstellung des Typs $Int \to Int \to Int$

keine Graphdarstellung, sondern einen Baum. Um es dem Benutzer weiterhin zu ermöglichen Ausdrücke mit Typannotationen und Typdefinitionen anzugeben, benötigen wir einen Algorithmus, der den Syntaxbaum in einen Graphen überführt.

Vom Baum zum Graph

Wie wir wissen treten äquivalente Unterbäume im Graph nur einmal auf. Die Idee für einen Algorithmus ergibt sich aus dieser Anforderung recht einfach. Wenn man einen Knoten für einen Unterbaum erzeugt, speichert man die Information ⟨Knoten,Unterbaum⟩. Für jeden Unterbaum, der im weiteren Verlauf auftritt, läßt sich mit $\equiv_\beta$ überprüfen, ob für ihn schon ein Knoten existiert. Ist dies der Fall, so verwendet man den vorhandenen Knoten. Ansonsten erzeugt man einen neuen.

Vom Graph zum Baum

Um den Graphen mit dem PrettyPrinter ausgeben zu können, benötigen wir ein Verfahren, um Graphen in Bäume zu transformieren. Diese Richtung birgt eine kleine Schwierigkeit. Wir haben mit Absicht nicht gefordert, daß der Graph azyklisch ist und somit erreicht, daß equi-rekursive Typen dargestellt werden können. Beispielsweise wird der Typ $\mu X.X \to Int$ durch den Graph in Abbildung 6 repräsentiert.

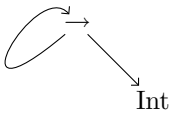


Abbildung 6: Graphdarstellung des Typs $\mu X.X \to Int$

Die Algorithmen müssen deshalb so gebaut werden, daß sie mit zyklischen Graphen umgehen können und beim Traversieren des Graphen nicht divergieren. Daher merkt man sich die Knoten, die besucht wurden. Wird ein Knoten zweimal besucht, so hat man einen Zyklus gefunden. Ein Zyklus im Graphen kann im Syntaxbaum als μ-Typ dargestellt werden.

7.7 Die Implementierung von $\equiv_\beta$

Durch die vollständige β-Reduktion ist die Relation $\equiv_\beta$ hochgradig divergenzgefährdet. Dieses Problem läßt sich natürlich nicht umgehen, da ihm das fun-

damentale Halteproblem zugrunde liegt. Allerdings können wir es durch eine elegante Implementierung leicht entschärfen[7]. Hauptidee ist, daß man die vollständige β-Reduktion bedarfsgesteuert durchführt.
Der Algorithmus erhält zwei Ausdrücke E_1 und E_2 als Eingabe und durchläuft dann die folgenden Schritte:

1. Teste, ob E_1 und E_2 syntaktisch gleich sind. Wenn ja, gebe *true* zurück; wenn nein, gehe zu Schritt 2.
2. Reduziere E_1 und E_2 zu einer schwachen β-Normalform.
3. Wenn E_1' und E_2' das gleiche Wurzel-Symbol haben, dann gehe in Rekursion.
4. Ansonsten gebe $false$ zurück.

Als erstes terminiert offensichtlich der Vergleich syntaktisch gleicher Ausdrücke. Als zweites terminiert in einigen Fällen auch der Vergleich zwischen einem divergierenden und einem normalisierenden Ausdruck, also beispielsweise $\lambda\_ : T.\Omega \not\equiv_\beta 1 + 1$. Ausdrücken, die schon bei der schwachen β-Reduktion divergieren, ist jedoch der Algorithmus schutzlos ausgeliefert.
In der Implementierung wird keine Regel für die Zusicherung benötigt. Da die Ausdrücke ausgewertet werden, fällt diese einfach unter den Tisch. Dies stellt kein Problem dar. Bevor zwei Ausdrücke auf Äquivalenz getestet werden, haben sie die Typüberprüfung durchlaufen. Offensichtlich gilt, wenn $(E_1\ as\ E_2)$ wohlgetypt ist, daß $\Gamma \vdash E_1 : T$ und $T \equiv_\beta E_2$. Daraus ergibt sich die folgende Implikation:

$$E_1 \equiv_\beta E_2 \Rightarrow E_1\ as\ T_1 \equiv_\beta E_2\ as\ T_2$$

Die Regel E-Refl wurde explizit angegeben, also für jedes einzelne Konstrukt eine extra Regel. Unter anderem wurde dabei folgende Regel verwendet:

$$\frac{x = y}{x \equiv_\beta y} \quad \text{(Eq-Var)}$$

Vollständig geparste Ausdrücke enthalten keine freien Variablen mehr. Tauchen dennoch zwei Variablen im Äquivalenztest auf, so kann es sich nur um zwei gebundene Bezeichner handeln.

> **Besonderheit wegen de Bruijn**
> Zwei gebundene Variablen sind genau dann äquivalent, wenn sie den gleichen de Bruijn Index haben. An dieser Stelle wäre die Verwendung expliziter Namen deutlich komplizierter, da die Ausdrücke modulo α-Umbenennung überprüft werden müßten.

Der Äquivalenztest kann im Interpreter auch separat als Kommando `equiv E1,E2` aufgerufen werden.

[7] Eine formale Betrachtung kann man in *A compiled implementation of string reduction* [6] finden

7.8 Implementierung von Referenzen

Referenzen wurden in LC^x aufgenommen, weil dadurch in F und F_ω interessante Kodierungen für imperative Objekte gebaut werden können. Da sich die Motivation auf einen sehr eingeschränkten Einsatzzweck bezog, sollte die übrige Architektur sowenig wie möglich verändert werden. Ein Durchfädeln des Speichers Σ durch die Auswertungsprozedur wäre nicht akzeptabel gewesen. Daher arbeitet *Cubint* mit einem globalen Speicher, der im Modul *Store* implementiert ist. Die Auswertungsregeln für Zuweisung und Dereferenzierung greifen einfach auf den globalen Speicher zu. Die Ergänzung um Referenzen erforderte deshalb nur eine lokale Änderung der Auswertungsprozedur.

8 *Cubint* in der Lehre

Cubint wurde für den Einsatz in der Lehre entwickelt. Konkret soll das Tool in der nächsten *Semantik*-Vorlesung an der Universität des Saarlandes (Fachbereich Informatik) eingesetzt werden. Dieser Abschnitt behandelt die unterschiedlichen Anwendungsmöglichkeiten für den Interpreter.

8.1 Benutzung

Der Interpreter soll schwerpunktmäßig zum Ausprobieren von Beispielen verwendet werden. Eine umfangreiche Beispielsammlung ist im Archiv von *Cubint* verfügbar. Mit Sicherheit macht es Sinn, diese am Anfang der Vorlesung nicht vollständig zugänglich zu machen, da sie als Referenz für Übungsaufgaben dienen kann.
Typischerweise beginnt die Vorlesung mit dem ungetypten λ-Kalkül. Auf diesem Übungsblatt könnte die erste Aufgabe lauten:

> **Aufgabe 1.1**
> *Machen Sie sich mit dem Interpreter* Cubint *vertraut, so daß sie die folgenden Aufgaben direkt am Interpreter überprüfen können. Das Handbuch,welches auf der Webseite verfügbar ist, bietet Informationen über die Installation des Interpreters, sowie eine vollständige Übersicht über alle Kommandos und die Syntax der Ausdrücke.*

Im untyped Modus des Interpreters wird nie ein Fehler gemeldet, wenn unsinnige Ausdrücke eingegeben werden. Man erhält *stuck terms* mit denen die Auswertungsfunktion nichts mehr anfangen kann. Zur Motivation von Typsystemen ist es daher interessant, ein paar einfache Beispiele solcher Ausdrücke auszuprobieren.

> **Aufgabe 1.2**
> *Gegeben sei der Ausdruck*
>
> ```
> def f = \x.\y.\z. if x then y else z .
> ```
>
> *Geben Sie zwei Kontexte für f an, die nicht vollständige ausgewertet werden können.*

So findet man schnell heraus, daß die Einführung von Zahlen und Operationen, die nur auf Zahlen funktionieren, einen neuen Mechanismus erfordern, der sicherstellt, daß die Auswertung nicht mittendrin hängenbleibt. In Verbindung mit der Beispielsammlung sollte es nach kurzer Einarbeitungszeit möglich sein, einen Großteil der Beispiele aus der Vorlesung am Interpreter nachzuvollziehen.

8.2 Eigene Erweiterung von *Cubint*

Ein eigener Type-Checker und ein eigener Evaluator können problemlos in den Interpreter integriert werden[8]. Für diesen Zweck wurde das Modul *UserCheckEval* angelegt. Der Programmierer muß sich als erstes mit der internen Syntax vertraut machen. Diese ist in der Datei *AbstractSyntax.sml* vollständig aufgelistet. Jedes Konstrukt ist mit einem *info* Feld ausgestattet, um die Stringdarstellung gebundener Namen für die Ausgabe zu erhalten. Es müssen die beiden Prozeduren *check* und *eval* nach der Signatur aus *UserCheckEval.sig* implementiert werden. In der Datei ist schon eine kleine Beispielimplementierung angegeben. Am Ende kann man den modifizierten Interpreter mit `make` neu compilieren. In den eigenen Modus kann man mit `switch tm ud` (für *user defined*) wechseln, oder man ruft den Interpreter mit `./cubint -ud` auf. *Cubint* kann wie gewohnt verwendet werden.

9 Fazit

Mit *Cubint* wurde ein Interpreter entwickelt, der Studenten beim Studium getypter λ-Kalküle unterstützen soll. Im Unterschied zu existierenden Implementierungen, wie beispielsweise denjenigen von Pierce [9], wurde als unterliegender Kalkül der λ-Cube gewählt. Er wurde systematisch um in Programmiersprachen übliche Konstrukte erweitert. Seine kompakte Syntax ermöglicht eine kompakte Darstellung der Algorithmen. Es gibt nur einen einzigen Typ-Checker, welcher die Typüberprüfung für die Kalküle S, F, F_ω und CC vornimmt und sowohl Terme als auch Typen auf Wohlgetyptheit testet. Auch die Auswertungsprozedur funktioniert für Terme und Typen, wodurch sich *Cubint* durch ein hohes Maß an Uniformität auszeichnet. Der λ-Cube scheint sich also als Grundlage für elegante Implementierungen zu eignen.
Daraus läßt sich allerdings nicht schließen, daß der Kalkül für praktische Zwecke, also als Basis für ein großes Programmiersystem, das System der Wahl ist. Im λ-Cube ist es erst durch das Typsystem möglich, Konzepte wie Terme, Typen und Kinds zu trennen. Es ist nicht leicht zu überblicken, welche Auswirkungen und welchen Mächtigkeitszuwachs die Integration neuer Konstrukte nachsichzieht. Auch ist fraglich, ob man nicht ein wesentliches Prinzip des Softwareentwurfs, nämlich *Low Coupling*, durch die Uniformität verletzt. Speziell bei komplexen Systemen ist eine klare Trennung von Konzepten zum frühstmöglichen Zeitpunkt wünschenswert.

Ziel dieser Arbeit war es, ein praktisch verwendbares System zu entwickeln, in dem sich S, F und F_ω wie gewohnt verhalten. Ein alternativer Ansatzpunkt

[8] Natürlich nur unter der Voraussetzung, daß man ML programmieren kann.
[9] http://www.cis.upenn.edu/~bcpierce/tapl/checkers

wäre es, die Erweiterungen in allgemeinst möglichen Form in den λ-Cube einzubetten. Dadurch gelangt man zum Beispiel automatisch zu der Frage, was sich mit Rekursion auf der Typebene ausdrücken läßt. Die Integration von Subtyping scheint ebenfalls ein großer Schritt zu sein und es bleibt zu untersuchen, ob und wie sich weiterführenden Aspekte, wie *bounded quantification*, überhaupt in den λ-Cube einbetten lassen. Viele Fragen, mit durchaus interessanten Forschungsaspekten, sind also in der Zukunft noch zu untersuchen.

Literatur

[1] Henk Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science, Volumes 1 (Background: Mathematical Structures) and 2 (Background: Computational Structures), Abramsky & Gabbay & Maibaum (Eds.), Clarendon*, volume 2. 1992.

[2] Henk Barendregt and Herman Geuvers. Proof-assistants using dependent type systems. In *Handbook of Automated Reasoning*, pages 1149–1238. 2001.

[3] Luca Cardelli. Amber. In Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet, editors, *Combinators and Functional Programming Languages*, pages 21–47. Springer-Verlag, 1986. Lecture Notes in Computer Science No. 242.

[4] Thierry Coquand and Gerard Huet. The calculus of constructions. *Inf. Comput.*, 76(2-3):95–120, 1988.

[5] Luis Damas and Robin Milner. Principal type schemes for functional programs. pages 207–212, 1982.

[6] Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction.

[7] S. Jones and E. Meijer. Henk: a typed intermediate language, 1997.

[8] Didier Le Botlan and Didier Rémy. MLF: Raising ML to the power of System-F. In *Proceedings of the International Conference on Functional Programming (ICFP 2003), Uppsala, Sweden*, pages 27–38. ACM Press, aug 2003.

[9] Lennart and Augustsson. Cayenne - a language with dependent types. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 239–250. ACM Press, 1998.

[10] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Trans. Programming Languages and Systems*, 4(2):258–282, 1982.

[11] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, Massachusetts;London, England, 2002.