Semantik eines verzögert auswertenden Lambdakalküls mit McCarthy's amb für Programmäquivalenz

David Sabel

Institut für Informatik
Fachbereich Informatik und Mathematik
Goethe-Universität Frankfurt am Main
sabel@ki.informatik.uni-frankfurt.de

Abstract: In diesem Beitrag werden die Ergebnisse der Untersuchung eines verzögert-auswertenden Lambdakalküls höherer Ordnung mit case-Ausdrücken, rekursivem letrec, einem seq-Operator und einem nichtdeterministischen Divergenz-vermeidenden Operator amb dargestellt. Als Gleichheitsbegriff wird kontextuelle Äquivalenz bzgl. einer May- und Must-Konvergenz verwendet. Mithilfe syntaktischer Methoden wird die Korrektheit von Programmtransformationen nachgewiesen und die Korrektheit von Übersetzungen untersucht. U.a. werden ein Kontextlemma, ein Standardisierungstheorem und die Gültigkeit einer endlichen Simulation gezeigt.

1 Einleitung

Computersysteme sind in vielen Bereichen der Industrie, Forschung und des alltäglichen Lebens nicht mehr wegzudenken. Im Gegenteil, in Zukunft werden weitere Anwendungsgebiete durch Rechnersysteme beeinflusst werden. Dieser Trend erhöht die Anforderung an die Informatik, mehr Verantwortung für korrekte Systeme zu übernehmen und fehlerhafte Systeme zu vermeiden. Während im Hardwareentwurf die Verifikation von Schaltungen regelmäßig durchgeführt wird, steckt die Softwareverifikation noch in den Kinderschuhen. In diesem Beitrag wird eine Übersicht über die Untersuchung aus [Sab08] gegeben, die sich mit der Korrektheit von Transformationen in Kompilern befasst. Offensichtlich ist Kompilerkorrektheit Grundvoraussetzung, um korrekte Software zu erstellen.

Ein Kompiler ist korrekt, wenn er die Bedeutung von Programmen nicht verändert. Der Begriff der Bedeutung benötigt hierbei eine einheitliche formale Notation, die möglichst kanonisch und auf viele Programmiersprachen anwendbar sein sollte. Die Informatik benutzt hierfür eine *formale Semantik*, d.h. ein korrekter Kompiler führt auf dem Weg von der Quell- zur Zielsprache nur Transformationen durch, welche die Semantik erhalten.

Solche Transformationen lassen sich grob in zwei Klassen einteilen. Zum einen gibt es Transformationen, die innerhalb einer Sprache durchgeführt werden. Dies sind z.B. Optimierungen, die unnötigen Code entfernen, oder Prozedur-Inlining, welches den Rumpf von Prozeduren einsetzt. Zum anderen gibt es Transformationen, die (höhere) Sprachen in andere (weniger höhere) Sprachen übersetzen, z.B. die Maschinenkodegenerierung.

Während es für die erste Form ausreichend ist, nachzuweisen, dass die Transformation die Programmäquivalenz innerhalb einer Sprache erhält, müssen für Übersetzungen zwei Sprachen und daher auch zwei unterschiedliche Semantiken verglichen werden. Beide Arten von Transformationen werden im folgenden untersucht und Methoden und Techniken dargestellt, die es erlauben deren Korrektheit nachzuweisen.

Alle modernen Programmiersprachen verfügen über Programmierkonstrukte, um Multi-Tasking- und Multi-User-Systeme zu implementieren, d.h. um Nebenläufigkeitsprimitive, die es ermöglichen mehrere Programme (Threads) quasi-parallel auszuführen. Durch die Popularität von Multi-Core-Architekturen erhöht sich der Bedarf an solchen Programmierprimitiven. Aus diesen Gründen sind Programmiersprachen zur nebenläufigen bzw. parallelen Programmierung von besonderem Interesse. In den verbleibenden Abschnitten dieses Beitrags wird über die Untersuchung eines Modells einer nebenläufigen (funktionalen) Programmiersprache hinsichtlich der Korrektheit von Transformationen berichtet.

Im Abschnitt 2 wird der $\Lambda^{\texttt{let}}_{\texttt{amb}}$-Kalkül dargestellt. Anschließend wird im Abschnitt 3 die kontextuelle Gleichheit von Programmen erörtert. Im Abschnitt 4 wird erläutert, wie Korrektheit von Programmtransformationen nachgewiesen werden kann, und welche Korrektheiten für den $\Lambda^{\texttt{let}}_{\texttt{amb}}$-Kalkül nachgewiesen wurden. Im Abschnitt 5 wird ein knapper Überblick über die Korrektheit von Übersetzungen und den erzielten Ergebnissen gegeben. Im Abschnitt 6 ist ein Fazit und ein Ausblick zu finden.

2 Ein Modell einer nebenläufigen funktionale Programmiersprache

Ein Modell für Programmiersprachen ist der von Church eingeführte Lambdakalkül [Chu41]. Darauf aufbauend wird im folgenden der $\Lambda^{\texttt{let}}_{\texttt{amb}}$-Kalkül definiert. Dieser ist ähnlich zum von Moran untersuchten call-by-need Kalkül [Mor98] und erweitert den Lambdakalkül um Konstrukte realistischer Programmiersprachen und insbesondere um den nichtdeterministischen `amb`-Operator. Obwohl dieser bereits 1963 von McCarthy [McC63] vorgeschlagene Operator keine echte Nebenläufigkeitsprimitive darstellt (im Sinne von Synchronisation und Datenaustausch zwischen Threads), reicht seine Ausdruckskraft aus, um nebenläufige Auswertung zu modellieren.

Sei Var eine Menge von unendlich vielen Variablen, wobei x, y, z im folgenden für Variablen stehen. Außerdem sei $\{T_1, \ldots, T_n\}$ eine Menge von Typen und für jeden Typ $D(T_i) = \{c_{T_i,1}, \ldots, c_{T_i,n_i}\}$ seine Menge von Datenkonstruktoren, wobei jedem $c_{T_i,j}$ eine Stelligkeit $\mathrm{ar}(c_{T_i,j}) \in \mathbb{N}_0$ zugeordnet sei. Z.B. ist der Typ `Bool` ein solcher Typ mit den Konstruktoren `True` und `False` (beide mit Stelligkeit 0), oder der Typ `Liste` mit den Konstruktoren `Cons` (Stelligkeit 2) und `Nil` (Stelligkeit 0). Die Syntax des $\Lambda^{\texttt{let}}_{\texttt{amb}}$-Kalküls ist durch folgende Grammatik beschrieben:

$$
\begin{aligned}
&s, t \in Exp ::= x \mid (\lambda x.s) \mid (s\ t) \mid (\texttt{seq}\ s\ t) \mid (\texttt{amb}\ s\ t) \mid (c_{T,i}\ s_1 \ldots s_{\mathrm{ar}(s_{T,i})}) \\
&\quad \mid (\texttt{letrec}\ x_1 = s_1, \ldots, x_m = s_m\ \texttt{in}\ s) \mid (\texttt{case}_{T_i}\ s\ Alt_{T_i,1} \ldots Alt_{T_i,|D(T_i)|}) \\
&Alt_{T_i,i} ::= (c_{T_i,i}\ x_{i,1} \ldots x_{i,\mathrm{ar}(c_{T_i,i})}) \to s
\end{aligned}
$$

D.h. die Sprache verfügt neben Abstraktion und Applikation, über letrec-Ausdrücke, die

rekursive Bindungen, sowie das Sharing von Ausdrücken erlauben, um Konstruktoranwendungen und case-Ausdrücke, um seq-Ausdrücke zur sequentiellen Auswertung und um den nichtdeterministischen amb-Operator. Für `amb` werden für seine beiden Argumente nebenläufige Auswertungen begonnen und der erste erhaltene Wert als Gesamtresultat übernommen. Durch diese Anforderungen an die Auswertung modelliert der Kalkül wesentliche Eigenschaften nebenläufiger Programmiersprachen.

Die Kombination des erweiterten Kalküls mit dem `amb`-Operator zeichnet sich auch dadurch aus, dass viele andere nichtdeterministische Operatoren damit kodiert werden können, z.B. ein Divergenz-vermeidendes `merge` zum nichtdeterministischen Mischen von Strömen, ein paralleler Konvergenztester `pconv`, ein Operator `choice` für die nichtdeterministische Auswahl, und das parallele Oder `por`. Die Implementierungen für die beiden letztgenannten Operatoren sind:

$$\begin{array}{rcl} \texttt{choice} & \equiv & \lambda x.\lambda y.((\texttt{amb}\ (\lambda z_1.x)\ (\lambda z_2.y))\ \texttt{True}) \\ \texttt{por} & \equiv & \lambda x.\lambda y.(\texttt{amb}\ (\texttt{if}\ x\ \texttt{then True else}\ y)\ (\texttt{if}\ y\ \texttt{then True else}\ x)) \end{array}$$

Im Bereich der formalen Semantik unterscheidet die Informatik im Wesentlichen drei Ansätze [Win93]: *Axiomatische Semantiken* modellieren Programmeigenschaften mit logischen Axiomen und erlauben die Herleitung weiterer Eigenschaften mithilfe logischer Schlussregeln. *Denotationale Semantiken* bilden Programme auf mathematische Objekte ab. Für funktionale Programmkalküle werden hierfür meistens so genannte Domains, d.h. partiell geordnete Mengen, verwendet. *Operationale Semantiken* definieren die Auswertung von Programmen. Dafür werden verschiedene Formalismen verwendet, z.B. Zustandübergangssysteme, abstrakte Maschinen und Ersetzungs- bzw Reduktionssysteme.

Axiomatische Semantiken werden eher zum Beschreiben einiger aber nicht sämtlicher Programmeigenschaften verwendet. Für `amb` funktionieren bisherige denotationale Ansätze nicht, z.B. da sich der Operator nicht monoton bzgl. gebräuchlicher Domain-Ordnungen verhält. Deshalb wird im folgenden eine operationale Semantik verwendet.

In Abb. 1 sind die Reduktionsregeln des $\Lambda^{\texttt{let}}_{\texttt{amb}}$-Kalküls aufgelistet. Die Regeln verwenden *Werte*, dies sind Abstraktionen und Konstruktoranwendungen, und Kontexte, wobei ein Kontext C ein Ausdruck ist, der an einer Position anstelle eines Unterausdrucks ein Loch besitzt. Das Einsetzen eines Ausdrucks s in das Loch eines Kontexts C wird als $C[s]$ geschrieben. Die Reduktionsregeln realisieren die verzögerte Auswertung, wie sie z.B. in Kompilern der Programmiersprache Haskell verwendet wird. Z.B. wird anstelle der vom Lambda-Kalkül bekannten (β)-Reduktion, die das Argument in den Rumpf einer Abstraktion voll einsetzt, die (lbeta)-Reduktion verwendet, die das Einsetzen des Arguments mithilfe eines `letrec`-Ausdrucks verzögert. Kopiert werden nur Abstraktionen (mittels der (cp)-Reduktion), und zwar erst dann, wenn der entsprechende Wert benötigt wird.

Als Standardreduktion wird die *Normalordnungsreduktion* für den $\Lambda^{\texttt{let}}_{\texttt{amb}}$-Kalkül definiert. Diese binäre Relation $\xrightarrow{\mathbf{no}}$ reduziert Ausdrücke des Kalküls schrittweise, indem die Reduktionsregeln nur an bestimmten Positionen eines Ausdrucks (mithilfe von so genannten Reduktionskontexten) angewendet werden. Für die vollständige Definition der Normalordnungsreduktion sei aus Platzgründen auf [SSS08, Sab08] verwiesen. Für Ausdrücke, die kein `amb` enthalten, ist die Normalordnungsreduktion eindeutig. Für `amb`-Ausdrücke

(lbeta)	$((\lambda x.s)\ r) \to (\texttt{letrec}\ x = r\ \texttt{in}\ s)$
(cp)	$\texttt{letrec}\ \ldots x_1 = (\lambda x.s), x_2 = x_1, \ldots, x_m = x_{m-1} \ldots C[x_m] \ldots$ $\to \texttt{letrec}\ \ldots x_1 = (\lambda x.s), x_2 = x_1, \ldots, x_m = x_{m-1} \ldots C[\lambda x.s] \ldots$
(llet-in)	$\texttt{letrec}\ E_1\ \texttt{in}\ (\texttt{letrec}\ E_2\ \texttt{in}\ r) \to \texttt{letrec}\ E_1, E_2\ \texttt{in}\ r$
(llet-e)	$\texttt{letrec}\ E_1, x = (\texttt{letrec}\ E_2\ \texttt{in}\ s)\ \texttt{in}\ r \to \texttt{letrec}\ E_1, E_2, x = s\ \texttt{in}\ r$
(lapp)	$(\texttt{letrec}\ E\ \texttt{in}\ t)\ s \to \texttt{letrec}\ E\ \texttt{in}\ (t\ s)$
(lcase)	$\texttt{case}_T\ (\texttt{letrec}\ E\ \texttt{in}\ t)\ alts \to \texttt{letrec}\ E\ \texttt{in}\ \texttt{case}_T\ t\ alts$
(lseq)	$\texttt{seq}\ (\texttt{letrec}\ E\ \texttt{in}\ s)\ t \to \texttt{letrec}\ E\ \texttt{in}\ \texttt{seq}\ s\ t$
(lamb-l)	$\texttt{amb}\ (\texttt{letrec}\ E\ \texttt{in}\ s)\ t \to \texttt{letrec}\ E\ \texttt{in}\ \texttt{amb}\ s\ t$
(lamb-r)	$\texttt{amb}\ s\ (\texttt{letrec}\ E\ \texttt{in}\ t) \to \texttt{letrec}\ E\ \texttt{in}\ \texttt{amb}\ s\ t$
(seq)	$\texttt{seq}\ v\ t \to t$, wenn v ein Wert ist, sowie analoge Fälle für $\texttt{seq}\ x\ t$, wobei x an einen Wert gebunden ist.
(case)	$\texttt{case}_T\ (c_{T,j}\ t_1 \ldots t_{\mathrm{ar}(c_{T,j})})\ \ldots ((c_{T,j}\ y_1 \ldots y_{\mathrm{ar}(c_{T,j})}) \to t) \ldots$ $\to \texttt{letrec}\ y_1 = t_1, \ldots, y_{\mathrm{ar}(c_{T,j})} = t_{\mathrm{ar}(c_{T,j})}\ \texttt{in}\ t$ sowie analoge Fälle für $\texttt{case}\ x\ \ldots$, wobei x an eine passende Konstruktoranwendung gebunden ist
(amb-l)	$(\texttt{amb}\ v\ s) \to v$, wenn v ein Wert ist, sowie analoge Fälle für $\texttt{amb}\ x\ s$, wobei x an einen Wert gebunden ist.
(amb-r)	$(\texttt{amb}\ s\ v) \to v$, wenn v ein Wert ist, sowie analoge Fälle für $\texttt{amb}\ s\ x$, wobei x an einen Wert gebunden ist.

Abbildung 1: Reduktionsregeln

ist sie jedoch nichtdeterministisch: Wenn Normalordnungsreduktionen innerhalb von s und innerhalb von t im Ausdruck $(\texttt{amb}\ s\ t)$ durchgeführt werden können, dann wählt die Normalordungsreduktion nichtdeterministisch eine der Möglichkeiten. Dies entspricht gerade der quasi-parallelen Auswertung der Argumente. Ebenso wird nichtdeterministisch eine der beiden Reduktionsregeln (amb-l) und (amb-r) gewählt, falls beide Argumente des amb-Ausdrucks Werte sind. Die Auswertung besteht nun aus dem wiederholten Anwenden von Normalordnungsreduktionen. Sie endet erfolgreich, sobald der auszuwertende Ausdruck in eine *schwache Kopfnormalform* (WHNF) überführt wurde. Dies sind Werte oder Ausdrücke der Form $\texttt{letrec}\ x_1 = s_1 \ldots\ x_n = s_n\ \texttt{in}\ v$ wobei v ein Wert oder eine Variable x_i ist, wobei x_i in der Umgebung an eine Konstruktoranwendung gebunden ist.

Die Normalordnungsreduktion achtet nicht auf eine *faire* Auswertung der verschiedenen Reduktionsmöglichkeiten. Tatsächlich ist der amb-Operator so auch nicht Divergenzvermeidend im operationalen Sinne: Sei $\bot$ ein Ausdruck der beliebig lange reduziert, ohne jemals einen Wert zu erreichen. Der Einfachheit halber sei $\bot \xrightarrow{\mathbf{no}} \bot$. Dann darf $\texttt{amb}\ \bot\ \texttt{True}$ stets auf sich selbst reduzieren: $\texttt{amb}\ \bot\ \texttt{True} \xrightarrow{\mathbf{no}} \texttt{amb}\ \bot\ \texttt{True} \xrightarrow{\mathbf{no}} \ldots$, obwohl das rechte Argument des amb-Ausdrucks bereits ein Wert ist.

Um diesem Problem zu begegnen, wird eine *faire Normalordnungsreduktion* $\xrightarrow{\mathbf{fno}}$ definiert: Hierfür werden sämtliche amb-Ausdrücke um zwei Ressourcen (nicht-negative Ganzzahlen) erweitert, d.h. anstelle von $(\texttt{amb}\ s\ t)$ wird $(\texttt{amb}_{\langle m,n \rangle}\ s\ t)$ verwendet. Jeder Normalordnungsschritt erniedrigt die zum Redex zugehörigen Ressourcen. Wenn beide

Ressourcen eines amb-Ausdrucks 0 sind, wird ein Scheduling durchgeführt, indem beide Ressourcen auf beliebige Ganzzahlen echt größer 0 erhöht werden. Unter Benutzung der fairen Normalordnung verhält sich der amb-Operator Divergenz-vermeidend. Für obiges Beispiel wird die Ressource für $\bot$ nach endlich vielen Schritten zu 0, so dass das rechte Argument gewählt werden muss.

3 Kontextuelle Äquivalenz und Beobachtungsbegriffe

Für Ausdrücke des $\Lambda^{\mathtt{let}}_{\mathtt{amb}}$-Kalküls wird ein Gleichheitsbegriff benötigt. Hierbei wird auf der Idee von Morris [Mor68] aufgebaut. Programme werden als gleich betrachtet, wenn sich ihr Verhalten bzgl. der Auswertung nicht unterscheiden lässt, wenn man die Programme als Unterprogramm eines beliebigen anderen Programms benutzt, d.h. wenn man sie in einen beliebigen Programmkontext einfügt. Es bleibt noch zu klären, welches Verhalten zu beobachten ist. Für deterministische Sprachen ist dies die Terminierung, die hier als *May-Konvergenz* bezeichnet sei: Ein Ausdruck s ist *may-konvergent* (geschrieben als $s{\downarrow}$), wenn s mithilfe von Normalordnungsreduktionsschritten zu einer schwachen Kopfnormalform reduziert werden kann, d.h. $s{\downarrow}$ genau dann, wenn $\exists \text{WHNF } t : s \xrightarrow{\mathbf{no},*} t$.

Für nichtdeterministische Sprachen reicht die Beobachtung der May-Konvergenz nicht aus, da dann Programme gleich sind, die unterschieden werden sollten, z.B. (choice True $\bot$) ist dann gleich zu True, obwohl der erste Ausdruck divergieren kann. Deswegen wird als zweiter Beobachungsbegriff die *Must-Konvergenz* betrachtet, wobei s *must-konvergent* ist ($s{\Downarrow}$), wenn jeder Nachfolger von s bezüglich beliebig vieler Normalordnungsreduktionen stets may-konvergent ist, d.h. $s{\Downarrow}$ genau dann, wenn $\forall t : s \xrightarrow{\mathbf{no},*} t \implies t{\downarrow}$. Das zu May-Konvergenz gegenteilige Prädikat wird *Must-Divergenz* ($s{\Uparrow}$) genannt und das zu Must-Konvergenz gegenteilige Prädikat wird als *May-Divergenz* ($s{\uparrow}$) bezeichnet.

Für die faire Normalordnungsreduktion können analoge Prädikate $\downarrow_F$ und $\Downarrow_F$ definiert werden, wobei zu Beginn der Auswertung sämtliche Ressourcen 0 betragen. Es zeigt sich jedoch, dass für viele Beweisführungen die Betrachtung der Normalordnungsreduktion ohne Fairnessbedingung ausreicht, denn es gilt:

Theorem 3.1. *$\downarrow = \downarrow_F$ und $\Downarrow = \Downarrow_F$.*

Während dieses Resultat für die May-Konvergenz nicht überrascht, ist es für die Must-Konvergenz eher unerwartet. Intuitiv bedeutet dies, dass die Definition der Must-Konvergenz bereits eine Art von eingebauter Fairness berücksichtigt.

Die kontextuelle Präordnung und Äquivalenz können nun definiert werden:

Definition 3.2. *Es sei $s \leq_c^{\downarrow} t$ genau dann, wenn $\forall C : C[s]{\downarrow} \implies C[t]{\downarrow}$ und $s \leq_c^{\Downarrow} t$ genau dann, wenn $\forall C : C[s]{\Downarrow} \implies C[t]{\Downarrow}$. Die* kontextuelle Präordnung *und die* kontextuelle Äquivalenz *sind definiert als $\leq_c := \leq_c^{\downarrow} \cap \leq_c^{\Downarrow}$ und $\sim_c := \leq_c \cap \geq_c$.*

Die kontextuelle Äquivalenz stellt im allgemeinen eine grobkörnige Gleichheit dar, d.h.

möglichst viele (sinnvolle) Gleichheiten sind in ihr enthalten. Sie ist eine Kongruenz, d.h. eine Äquivalenzrelation, die kompatibel mit Kontexten ist.

Eine Besonderheit des hier vorgestellten Ansatzes liegt in der Definition der Must-Konvergenz. Diese weicht von vorherigen Definitionen einer (totalen) Must-Konvergenz ab (wie sie z.B. in [Mor98] verwendet wird), da die Existenz einer unendlich langen Normalordnungsreduktionsfolge nicht ausreicht, um Must-Konvergenz zu widerlegen (und damit May-Divergenz zu beweisen). Vielmehr muss die Existenz einer endlichen Normalordnungsreduktionsfolge gezeigt werden, die in einem Ausdruck endet, der nicht mehr konvergieren kann (d.h. must-divergent ist). Der hier verwendete Begriff der Must-Konvergenz hat den Vorteil gegenüber der Verwendung der totalen Must-Konvergenz, dass echte fehlerhafte Ausdrücke von Ausdrücken unterschieden werden, die zwar unendlich lange auswerten können, aber die Möglichkeit zu terminieren nie verlieren. Das letztere Verhalten ist gerade typisch für reaktive Systeme, z.B. Betriebssysteme: Sie können unendlich lange laufen, aber stets durch ein Herunterfahren-Kommando beendet werden.

Eine Programmtransformation ist eine binäre Relation auf Ausdrücken.

Definition 3.3. *Ein Programmtransformation P ist* korrekt *genau dann, wenn $P \subseteq \sim_c$.*

4 Nachweis der Korrektheit von Programmtransformationen

Das Widerlegen der Korrektheit einer Programmtransformation P ist einfach. Es reicht aus, Ausdrücke s, t mit $(s, t) \in P$ und einen Kontext C anzugeben, der s und t bzgl. ihres Konvergenzverhaltens unterscheidet. Der Nachweis der Korrektheit ist hingegen schwierig, da gleiches Konvergenzverhalten für alle Paare $(s, t) \in P$ bzgl. aller (unendlich vielen) Programmkontexte C nachgewiesen werden muss.

Ein erster Schritt zur Vereinfachung ist ein so genanntes Kontextlemma, welches die zu betrachtenden Kontexte einschränkt. In [Mor98] wurde ein Kontextlemma bzgl. der May-Konvergenz nachgewiesen, allerdings wurde dort kein Beweis für den Must-Konvergenz-Teil angegeben. Aufgrund dessen beziehen sich die dortigen Korrektheitsresultate nur auf kontextuelle Gleichheit bzgl. der May-Konvergenz, was gerade in einem Kalkül mit `amb` nicht ausreichend ist. In [Sab08] konnte das folgende (vollständige) Kontextlemma nachgewiesen werden, wobei R Reduktionskontexte sind.

Lemma 4.1 (Kontextlemma)**.** *Für Ausdrücke s, t gilt*

$$(\forall R : (R[s]{\Downarrow} \implies R[t]{\Downarrow}) \wedge (R[s]{\downarrow} \implies R[t]{\downarrow})) \implies s \leq_c t$$

Im folgenden wird eine Technik zum Nachweis der Korrektheit von Programmtransformationen skizziert. Hierfür wird der Begriff der *Oberflächenkontexte* S benötigt: Für diese Kontexte darf die Position des Lochs nicht im Rumpf einer Abstraktion liegen. Jeder Reduktionskontext ist auch ein Oberflächenkontext. Um für eine Transformation P zu zeigen, dass $P \subseteq \leq_c$ gilt, wird deren Anwendung in Oberflächenkontexten betrachtet, d.h. für $(s, t) \in P$ die Transformation $S[s] \xrightarrow{S,P} S[t]$ für Oberflächenkontexte S. Aus-

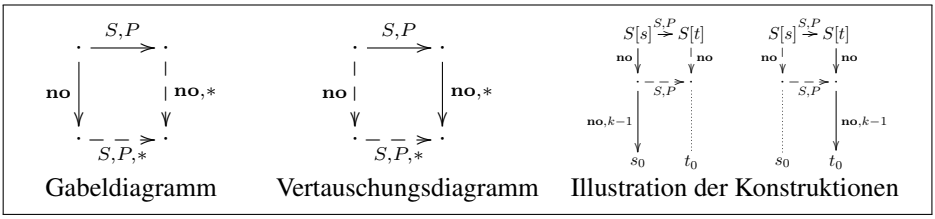

Abbildung 2: Gabel- und Vertauschungsdiagramme

gehend von erfolgreichen Reduktionsfolgen für $S[s]$ werden erfolgreiche Reduktionsfolgen für $S[t]$ konstruiert, was $\forall R : R[s]{\downarrow} \implies R[t]{\downarrow}$ impliziert. Ebenso werden für *fehlschlagende* Reduktionsfolgen für $S[t]$, d.h. Reduktionsfolgen der Form $S[t] \xrightarrow{\mathbf{no},*} t'$ mit $t'{\Uparrow}$, fehlschlagende Reduktionsfolgen für $S[s]$ konstruiert. Dies impliziert für alle $R : R[t]{\uparrow} \implies R[s]{\uparrow}$, was logisch äquivalent zu $\forall R : R[s]{\Downarrow} \implies R[t]{\Downarrow}$ ist. Da damit die Voraussetzungen für das Kontextlemma gezeigt sind, gilt $s \leq_c t$.

Die Konstruktion von Reduktionsfolgen benutzt *vollständige Mengen von Gabel- bzw. Vertauschungsdiagrammen*. Diese dienen zur vollständigen Darstellung aller Überlappungen einer Normalordnungsreduktion mit einer Transformationsregel sowie der Zusammenführbarkeit. Während Gabeldiagramme die Überlappungen der Form $s' \xleftarrow{\mathbf{no}} s \xrightarrow{S,P} t$ beschreiben, repräsentieren Vertauschungsdiagramme Überlappungen der Form $s \xrightarrow{S,P} t \xrightarrow{\mathbf{no}} t'$. Abb. 2 zeigt die Grundform von Gabel- und Vertauschungsdiagrammen, hierbei sind durchgezogene Pfeile gegebene Reduktionen und gestrichelte Pfeile zeigen die Zusammenführbarkeit. Mittels Induktion (im einfachsten Fall über die Länge der gegebenen Reduktionsfolge) und unter Verwendung der Diagramme, kann dann obig erwähnte Konstruktion erfolgen. Im rechten Teil von Abb. 2 ist die Konstruktion im Induktionsbeweis für den einfachsten Fall (alle Diagramme sind Quadrate) illustriert. Mithilfe dieser Technik wurde in [Sab08] gezeigt:

Theorem 4.2. *Alle Reduktionsregeln aus Abb. 1 außer* (amb-l)*,* (amb-r) *sind korrekt.*

Ferner wurden weitere Programmtransformationen, die häufig als Optimierungen in Kompilern eingesetzt werden, untersucht. Abb. 3 listet den Großteil davon auf. Hierbei sind Ω-Ausdrücke solche $\boldsymbol{\Lambda}^{\mathtt{let}}_{\mathtt{amb}}$-Terme s für die gilt: für alle Umgebungen E ist $\mathtt{letrec}\ E\ \mathtt{in}\ s$ must-divergent. In [Sab08] wurde gezeigt:

Theorem 4.3. *Die Transformationen aus Abb. 3 (in allen Kontexten) sind korrekt.*

Eine weitere wichtige Aussage wird im Standardisierungstheorem bewiesen:

Theorem 4.4 (Standardisierungstheorem)**.** *Wenn $t \xrightarrow{*} t'$, wobei t' eine WHNF ist und $\xrightarrow{*}$ korrekte Programmtransformationen und* (amb)*-Reduktionen in beliebigen Kontexten enthält, dann gilt $t{\downarrow}$. Wenn $t \xrightarrow{*} t'$, wobei $t'{\Uparrow}$ und $\xrightarrow{*}$ korrekte Programmtransformationen und* (amb)*-Reduktionen in Oberflächenkontexten enthält, dann gilt $t{\Uparrow}$.*

Mit diesem Resultat können Korrektheitsnachweise einfacher geführt werden, da nicht alleinig mit Normalordnungsreduktionsfolgen argumentiert werden muss, sondern auch

(gc1) $\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ t \to t$, falls kein x_i in t vorkommt.
(gc2) $\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n, E\ \texttt{in}\ t \to \texttt{letrec}\ E\ \texttt{in}\ t$,
falls kein x_i in t oder E vorkommt.
(cpx) $\texttt{letrec}\ x = y, \ldots C[x] \ldots \to \texttt{letrec}\ x = y, \ldots C[y] \ldots$, wenn $x, y \in \mathit{Var}$
(cpcx) $\texttt{letrec}\ x = (c\ s_1 \ldots s_n), \ldots C[x] \ldots$
$\to \texttt{letrec}\ x = (c\ y_1 \ldots y_n), y_1 = s_1, \ldots, y_n = s_n, \ldots C[(c\ y_1 \ldots y_n)] \ldots$
wobei y_i neue Variablen
(ucp) $\texttt{letrec}\ x = s, \ldots S[x]\ \ldots \to \texttt{letrec} \ldots S[s] \ldots$
falls S ein Oberflächenkontext und x nur an der ersetzten Position vorkommt
(cpom) $\texttt{letrec}\ x = s\ \ldots\ C[x]\ \ldots \to \texttt{letrec}\ x = s\ \ldots\ C[s]\ \ldots$,
falls s ein Ω-Ausdruck ist.

Abbildung 3: Weitere Programmtransformationen

andere Transformationsfolgen konstruiert werden können, für die stets eine Folge von Normalordnungsreduktionen existiert. Z.B. lässt sich damit nachweisen, dass alle Ω-Ausdrücke kontextuell gleich sind:

Theorem 4.5. *Für Ω-Terme s und t gilt $s \sim_c t$.*

Ein weiteres Kriterium ist eine Form der endlichen Simulation. Hierbei kann Gleichheit gefolgert werden, indem nur die Auswertung der zu untersuchenden Terme betrachtet wird, ohne jedoch die Auswertung in Kontexten zu berücksichtigen. Für einen Ausdruck s ist die vollständige Nachfolgermenge $\mathrm{sc}_i(s)$ der Tiefe i induktiv definiert:

$$\begin{aligned} \mathrm{sc}_0(s) &= \{s\} \\ \mathrm{sc}_i(s) &= \begin{cases} \{t_2 \mid t_1 \xrightarrow{\mathbf{no}} t_2, t_1 \in \mathrm{sc}_{i-1}(s)\}, \text{ wenn } \exists t : s \xrightarrow{\mathbf{no}} t \\ \mathrm{sc}_{i-1}(s), \text{ sonst} \end{cases} \end{aligned}$$

Der Begriff der kontextuellen Präordnung kann wie folgt auf Mengen von Ausdrücken erweitert werden als $\langle\leq\rangle := \langle\leq^{\downarrow}\rangle \cap \langle\leq^{\Downarrow}\rangle$, wobei $M\ \langle\leq^{\downarrow}\rangle\ N$ genau dann, wenn $\forall s \in M : \exists t \in N : s \leq_c^{\downarrow} t$ und $M\ \langle\leq^{\Downarrow}\rangle\ N$ genau dann, wenn $\forall t \in N : \exists s \in M : s \leq_c^{\Downarrow} t$. In [Sab08] wurde bewiesen:

Theorem 4.6. *Für geschlossene s, t und $i, j \in \mathbb{N}_0$ gilt:* $\mathrm{sc}_i(s)\ \langle\leq\rangle\ \mathrm{sc}_j(s) \implies s \leq_c t$.

Dieses Resultat erlaubt den Schluss auf kontextuelle Gleichheit, falls kontextuelle Mengengleichheit für zwei vollständige Nachfolgermengen gezeigt werden kann. Z.B. lässt sich damit sofort nachweisen, dass `amb True False` $\sim_c$ `amb False True` erfüllt ist, denn beide Ausdrücke haben $\{\texttt{True}, \texttt{False}\}$ als Nachfolgermenge. Mithilfe dieses Kriteriums und den anderen Beweismethoden wurden u.a. die Gleichheiten in Abb. 4 nachgewiesen, die charakteristische Eigenschaften von nichtdeterministischen Operatoren ausdrücken.

Eine Untersuchung der kontextuellen Präordnung zeigt, dass mithilfe von `amb` und dem Must-Konvergenz-Test auch auf May-Konvergenz getestet werden kann, was folgende etwas überraschende Charakterisierung der kontextuellen Gleichheit ermöglicht.

Theorem 4.7. *$s \sim_c t$ genau dann, wenn $C[s]{\Downarrow} \iff C[t]{\Downarrow}$.*

$(\texttt{amb}\ s\ t) \sim_c t$, wenn s ein Ω-Term ist.
$(\texttt{amb}\ v_1\ v_2) \sim_c (\texttt{choice}\ v_1\ v_2)$ für geschlossene Werte v_1, v_2
$(\texttt{amb}\ s\ t) \sim_c (\texttt{amb}\ t\ s)$ und $(\texttt{amb}\ v\ v) \sim_c v$ für einen Wert v
$(\texttt{amb}\ v_1\ (\texttt{amb}\ v_2\ v_3)) \sim_c (\texttt{amb}\ (\texttt{amb}\ v_1\ v_2)\ v_3)$ für geschlossene Werte v_i.
$(\texttt{choice}\ s\ t) \sim_c (\texttt{choice}\ t\ s)$ und $(\texttt{choice}\ s\ s) \sim_c s$ für geschlossene s, t.
$(\texttt{choice}\ s_1\ (\texttt{choice}\ s_2\ s_3)) \sim_c (\texttt{choice}\ (\texttt{choice}\ s_1\ s_2)\ s_3)$ für geschlossene s_i
$(\texttt{por}\ s\ \texttt{True}) \sim_c \texttt{True} \sim_c (\texttt{por}\ \texttt{True}\ s)$ und $(\texttt{por}\ \texttt{False}\ \texttt{False}) \sim_c \texttt{False}$

Abbildung 4: Charakteristische Gleichheiten nichtdeterministischer Operatoren

5 Korrektheit von Übersetzungen

Wie bereits erwähnt, gibt es eine weitere Klasse von Transformationen, die ein Kompiler durchführt, die Übersetzung einer Sprache L_1 in eine andere Sprache L_2. Im folgenden seien $\downarrow_i, \Downarrow_i$, $i = 1, 2$ die May- und Must-Konvergenzprädikate der beiden Sprachen.

Definition 5.1. *Eine Übersetzung $T : L_1 \to L_2$ ist* konvergenzäquivalent*, falls $s\downarrow_1 \iff T(s)\downarrow_2$ und $s\Downarrow_1 \iff T(s)\Downarrow_2$ für alle $s \in L_1$.*

Ist eine Übersetzung konvergenzäquivalent, so darf die Zielsprache zumindest zum Ausführen der Programme benutzt werden. Werden jedoch Transformationen innerhalb der Zielsprache durchgeführt, dann sollte die Übersetzung adäquat sein:

Definition 5.2. *Eine Übersetzung $T : L_1 \to L_2$ ist* adäquat*, falls für alle $s, t \in L_1$ stets gilt $T(s) \leq_{c,1} T(t) \implies s \leq_{c,1} t$. Sie ist* voll-abstrakt*, falls für alle $s, t \in L_1$ stets gilt $T(s) \leq_{c,1} T(t) \iff s \leq_{c,1} t$.*

Für eine genauere Betrachtung und Methoden um Adäquatheit zu zeigen sei auf [SSNSS08] verwiesen. Für den $\mathcal{A}^{\texttt{let}}_{\texttt{amb}}$-Kalkül wurde in [Sab08] eine abstrakte Maschine zur Auswertung und eine (mehrstufige) Übersetzung Υ der Kalkülsprache in die Maschinensprache definiert. Dafür wurde gezeigt:

Theorem 5.3. *Die Übersetzung Υ ist konvergenzäquivalent.*

Dies bedeutet, dass die Übersetzung in die Maschinensprache korrekt ist, oder auch umgekehrt, dass die Maschine korrekt spezifiziert wurde. Für eine Zwischenübersetzung konnte sogar volle Abstraktheit nachgewiesen werden.

6 Fazit und Ausblick

Aufbauend auf der operationalen Semantik wurde Korrektheit von Programmtransformationen bzgl. kontextueller Gleichheit für einen nichtdeterministischen Programmkalkül nachgewiesen. Die Verwendung von May- und Must-Konvergenz in der Gleichheitsdefinition führt zu erwarteten Gleichungen. Die präsentierten Techniken sind größtenteils

unabhängig vom konkreten Kalkül, so dass diese für weitere Forschung auch auf andere nebenläufige Kalküle übertragbar sein sollten. In [NSSSS07] wurde dies bereits für einen nebenläufigen call-by-value Prozesskalkül mit Futures und Speicherzellen bewerkstelligt.

Literatur

[Chu41] A. Church. *The Calculi of Lambda-Conversion*. Princeton University Press, Princeton, New Jersey, 1941.

[McC63] J. McCarthy. A Basis for a Mathematical Theory of Computation. *Computer Programming and Formal Systems*, Seiten 33–70. North-Holland, Amsterdam, 1963.

[Mor68] J. H. Morris. *Lambda Calculus Models of Programming Languages*. Dissertation, MIT, Cambridge, MA, 1968.

[Mor98] A. Moran. *Call-by-name, Call-by-need, and McCarthy's Amb*. Dissertation, Department of Computing Science, Chalmers University of Technology and University of Gothenburg, Sweden, 1998.

[NSSSS07] J. Niehren, D. Sabel, M. Schmidt-Schauß und J. Schwinghammer. Observational Semantics for a Concurrent Lambda Calculus with Reference Cells and Futures. *MFPS 2007*, *ENTCS*, 173:313–337, 2007.

[Sab08] D. Sabel. *Semantics of a Call-by-Need Lambda Calculus with McCarthy's amb for Program Equivalence*. Dissertation, Goethe-Universität Frankfurt am Main, Institut für Informatik, 2008. Veröffentlicht im Verlag Dr. Hut, ISBN 978-3-89963-866-0, http://www.ki.informatik.uni-frankfurt.de/papers/sabel/dissertation-sabel.pdf.

[SSNSS08] M. Schmidt-Schauß, J. Niehren, J. Schwinghammer und D. Sabel. Adequacy of Compositional Translations for Observational Semantics. *Fifth IFIP TCS 2008*, *IFIP* 273, Seiten 521–535. Springer, 2008.

[SSS08] D. Sabel und M. Schmidt-Schauß. A Call-by-Need Lambda-Calculus with Locally Bottom-Avoiding Choice: Context Lemma and Correctness of Transformations. *Math. Structures Comput. Sci.*, 18(3):501–553, 2008.

[Win93] G. Winskel. *The formal semantics of programming languages: an introduction*. MIT Press, Cambridge, MA, USA, 1993.

David Sabel wurde 1977 in Schotten geboren. Er studierte von 1997 bis 2003 Informatik im Diplomstudiengang mit dem Nebenfach Betriebswirtschaftlehre an der Goethe-Universität in Frankfurt am Main. In den Jahren 2004 bis 2008 promovierte er im Fach Informatik an der Goethe-Universität in Frankfurt am Main und beendete seine Promotion erfolgreich im Oktober 2008. Er war von Mai 2004 bis April 2009 als wissenschaftlicher Mitarbeiter an der Professur für Künstliche Intelligenz und Softwaretechnolgie am Institut für Informatik des Fachbereichs Informatik und Mathematik der Goethe-Universität Frankfurt am Main beschäftigt. Seit Mitte April 2009 ist er dort als Akademischer Rat tätig und forscht im Bereich der Programmiersprachensemantik.