Korrektheit von Programmen und Programmiersprachen
Methoden, Analysen und Anwendungen

Kumulative Habilitationsschrift
vorgelegt beim Fachbereich Informatik und Mathematik
der Johann Wolfgang Goethe – Universität
in Frankfurt am Main

von

David Sabel

aus Schotten

Frankfurt 2013

Danksagung

An dieser Stelle möchte ich mich bei all jenen ganz herzlich bedanken, die zu diesem Projekt beigetragen haben.

Mein ganz besonderer Dank gilt Prof. Dr. Manfred Schmidt-Schauß, der mich auf meinem langjährigen Weg meiner akademischen Ausbildung begleitet hat. Seine stetige Bereitschaft zu Diskussionen und Gesprächen, seine motivierenden Worte, sein stetiger Drang, Neues zu entdecken, seine Hartnäckigkeit, auch bei schwierigen Problemen nicht lockerzulassen, und schließlich auch die vielen netten Gespräche über Dies und Das sind von unschätzbarem Wert. Auch für die Erstellung eines Gutachtens zu meiner Habilitationsschrift möchte ich ihm an dieser Stelle ganz herzlich danken.

Ebenso bedanke mich bei den externen Gutachtern Herrn Prof. Dr. Martin Hofmann und Herrn Prof. Dr. Gert Smolka für die Übernahme und Erstellung der Gutachten.

Ich danke allen Beteiligten am Fachbereich Informatik und Mathematik der Goethe-Universität für ihre Unterstützung und den reibungslosen und zügigen Verlauf meines Habilitationsverfahrens. Insbesondere danke ich dabei den Mitgliedern meiner Habiliationskommission Prof. Dr. Alexander Mehler, Prof. Dr. Georg Schnitger, Prof. Dr. Thorsten Theobald und Prof. Dott.-Ing. Robert Zicari.

Ebenfalls möchte ich mich an dieser Stelle bei allen Koautoren für die spannende und erfolgreiche Zusammenarbeit bedanken. Insbesondere danke ich Dr. Joachim Niehren und Dr. Jan Schwinghammer für die interessante Zusammenarbeit, die gemeinsamen Forschungstreffen und die vielen ergiebigen Diskussionen. Bei Prof. Dr. Elena Machkasova möchte ich mich ebenfalls für die gemeinsame Zusammenarbeit während ihrer Forschungsaufenthalte in Frankfurt ganz herzlich bedanken.

Meinem Kollegen und Zimmergenossen Conrad Rau danke ich für die vielen Diskussionen, Plaudereien und die freundschaftliche Zusammenarbeit. Bei Angelika Schifignano und Marion Terrell bedanke ich mich für die netten Pläusche, die mir die notwendige kurze Ablenkung vom beruflichen Alltag geben.

Zu guter Letzt danke ich meiner Familie und meinen Freunden für ihre ständige Unterstützung und aufmunternden Worte. Insbesondere danke ich meinem Bruder Elmar und seiner Frau Simone, sowie meiner Schwester Alexandra und ihrem Mann Manfred.

David Sabel

Inhaltsverzeichnis

1 Einleitung

Informatische Systeme sind heutzutage aus dem alltäglichen Leben nicht mehr wegzudenken. Neben klassischen Anwendungen findet man Systeme in immer mehr Alltagsgegenständen wie z. B. Fernsehern, Kühlschränken, Spülmaschinen, etc. Nicht zuletzt durch die zunehmende Vernetzung von Systemen wird dieser Trend weiter anhalten.

Auch durch diese stetig anwachsende Verbreitung informatischer Systeme wird es immer wichtiger, fehlerhafte Systeme zu vermeiden, um durch Softwarefehler entstandene Schäden zu minimieren und die Verlässlichkeit von Programmen zu steigern. Dies trifft insbesondere für sicherheitsrelevante Systeme zu, deren Fehler zu unermesslichem Schaden führen können. Da die Informatik für diese Systeme (mit-)verantwortlich ist, trifft jeden Informatiker eine entsprechende Sorgfaltspflicht, korrekte Systeme zu entwickeln und Programmierfehler zu vermeiden.

Dabei stellt sich jedoch schon die grundlegende Frage, welche Anforderungen oder Eigenschaften die Korrektheit eines Systems, der Software oder eines Programms festlegen. Wurde eine solche Definition der „Korrektheit“ gefunden, so müssen Methoden entwickelt (und angewendet) werden, um diese für konkrete Programme und Programmiersprachen nachzuweisen. Neben Test-basierter Verifikation, die i. A. nur eine mehr oder weniger wahrscheinliche Korrektheit liefern kann, bietet das Gebiet der formalen Semantik die Möglichkeit, Korrektheit zu definieren und diese mathematisch nachzuweisen.

Obwohl die Semantik von Programmiersprachen ein gut untersuchter Zweig der Informatik ist, lassen die bisherigen Untersuchungen viele Fragen offen. Das Gebiet benötigt stetiges Erforschen neuer Methoden und Techniken, um sich den aktuellen Anforderungen und der Weiterentwicklung moderner Programmiersprachen anzupassen. Insbesondere ist hier die nebenläufige, parallele und verteilte Programmierung zu nennen. Zum einen erfordert die stetige Erhöhung der Anzahl an Prozessorkernen (Multicore-Architekturen und auch GPGPU-

Programmierung) neue (parallele) Programmiermodelle und -methodiken. Zum anderen erfordert die zunehmende Vernetzung von Systemen gleichzeitig die Programmierung verteilter Systeme.

Im Rahmen dieser Arbeit wird ein Beitrag zum Fernziel der Erschaffung korrekter und fehlerfreier Software geleistet. Zum einen wird die Korrektheit von Programmiersprachen selbst, zum anderen die Korrektheit von Implementierungen, d. h. Programmen, untersucht, analysiert und schließlich auch für konkrete Modelle insbesondere funktionaler und nebenläufiger Programmiersprachen nachgewiesen. Zusätzlich werden auch Eigenschaften und Vergleiche von Programmiersprachen, die letztlich zu einem genaueren Verständnis der Programmiersprachensemantik führen, untersucht. Dabei werden je nach Anwendungsfall verschiedene Korrektheitsbegriffe erörtert und untersucht. Beispielsweise ist Korrektheit anders zu definieren für Transformationen von Programmen innerhalb einer Programmiersprache, als die Korrektheit einer Übersetzung (wie sie z. B. während des Compilierens durchgeführt wird) einer höheren Programmiersprache in eine maschinennähere Sprache. Will man einzelne Aussagen bzw. Eigenschaften über Programme als korrekt nachweisen, so bietet sich die Verwendung einer Logik an, wobei die Theoreme der Logik als „korrekte Aussagen“ angesehen werden. Die in dieser Zusammenfassung dargestellten Resultate umfassen all diese verschiedenen Problemstellungen.

Allen im Folgenden vorgestellten Resultaten ist gemeinsam, dass sie direkt auf der operationalen Semantik der Programmiersprachen aufsetzen und den darauf aufbauenden, natürlichen Gleichheitsbegriff der kontextuellen Äquivalenz verwenden.

Die operationale Semantik einer Programmiersprache legt fest, wie Programme ausgeführt bzw. ausgewertet werden sollen, d. h. durch Anwenden der operationalen Semantik kann das Resultat der Ausführung eines Programms ermittelt werden. Im Grunde definiert die operationale Semantik daher mehr oder weniger den Interpreter der Programmiersprache. Je nach Art der Programmiersprache kann das Resultat der Ausführung ein veränderter Zustand des Systems oder ausschließlich ein Wert sein. Die erste Variante ist in imperativen Programmiersprachen vorzufinden, deren Programme daher semantisch als Manipulation des Systemzustands zu interpretieren sind. In reinen funktionalen Programmiersprachen hingegen sind Seiteneffekte – d. h. sichtbare Speicheränderungen – unzulässig und daher legt die operationale Semantik als Resultat der Programmausführung den Wert des Programms fest. Daher spricht man in diesen Programmiersprachen eher von der Auswertung anstelle der Ausführung eines Programms. Allgemein gibt es verschiedene Ansätze zur Definition der operationalen Semantik einer Programmiersprachen. Beispielsweise kann diese als Zustandsübergangssystem, als Ersetzungssystem auf Termen und als abstrakte Maschine angegeben werden.

Der Gleichheitsbegriff der kontextuellen Äquivalenz baut direkt auf der operationalen Semantik der Programmiersprache auf. Die kontextuelle Äquivalenz identifiziert Programme dann als gleich, wenn man sie in allen beliebigen Kontexten (d. h. beliebigen umgebenden größeren Programmen) durch Auswerten – d. h. bzgl. der operationalen Semantik – nicht unterscheiden kann. Dieser Begriff ist sehr natürlich, da der Austausch eines (Unter-)Programms durch ein kontextuell gleiches Unterprogramm nicht beobachtet werden kann. Für den Begriff

der Beobachtbarkeit ist dabei die Terminierung ausreichend, wobei für nichtdeterministische (z. B. nebenläufige) Programmiersprachen sowohl beobachtet werden muss, ob die Möglichkeit zur Terminierung aber auch die Möglichkeit zur Nichtterminierung besteht.

1.1 Überblick

Grundlage dieser kumulativen Habilitationsschrift sind die folgenden Veröffentlichungen[1]:

1. M. Schmidt-Schauß, J. Niehren, J. Schwinghammer & D. Sabel
   Adequacy of compositional translations for observational semantics.
   In *Fifth IFIP International Conference On Theoretical Computer Science - TCS 2008, IFIP 20th World Computer Congress, TC 1, Foundations of Computer Science, September 7-10, 2008, Milano, Italy*, Bd. 273 von *IFIP*, S. 521–535. Springer, 2008.

   DOI: 10.1007/978-0-387-09680-3_35

2. J. Schwinghammer, D. Sabel, M. Schmidt-Schauß & J. Niehren
   Correctly translating concurrency primitives.
   In *ML '09: Proceedings of the 2009 ACM SIGPLAN workshop on ML*, S. 27–38. ACM, New York, NY, USA, 2009.

   DOI: 10.1145/1596627.1596633

3. D. Sabel, M. Schmidt-Schauß & F. Harwath
   Reasoning about contextual equivalence: From untyped to polymorphically typed calculi.
   In *INFORMATIK 2009, Im Focus das Leben, Beiträge der 39. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 28.9 - 2.10.2009 in Lübeck*, Bd. 154 von *GI Edition - Lecture Notes in Informatics*, S. 369; 2931–45, (4. Arbeitstagung Programmiersprachen (ATPS)). Köllen Druck+Verlag, Bonn, Germany, 2009.

   URL: http://subs.emis.de/LNI/Proceedings/Proceedings154/article2805.html

4. M. Schmidt-Schauß, D. Sabel & E. Machkasova
   Simulation in the call-by-need lambda-calculus with letrec.
   In *Proceedings of the 21st International Conference on Rewriting Techniques and Applications*, Bd. 6 von *Leibniz International Proceedings in Informatics (LIPIcs)*, S. 295–310. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2010.

   DOI: 10.4230/LIPIcs.RTA.2010.295

5. M. Schmidt-Schauß & D. Sabel
   On generic context lemmas for higher-order calculi with sharing.

[1] Aufgrund des Copyrights der Verlage, sind die Veröffentlichungen in dieser Version nicht angehängt. Jedoch sind diese über die angegeben DOIs bzw. URLs auffindbar

Theoretical Computer Science, 411(11-13):1521 – 1541, 2010.

DOI: 10.1016/j.tcs.2009.12.001

6. M. Schmidt-Schauß & D. Sabel
Closures of may-, should- and must-convergences for contextual equivalence.
Information Processing Letters, 110(6):232 – 235, 2010.

DOI: 10.1016/j.ipl.2010.01.001

7. M. Schmidt-Schauß, D. Sabel & E. Machkasova
Counterexamples to applicative simulation and extensionality in non-deterministic call-by-need lambda-calculi with letrec.
Information Processing Letters, 111(14):711–716, 2011.

DOI: 10.1016/j.ipl.2011.04.011

8. D. Sabel & M. Schmidt-Schauß
A contextual semantics for Concurrent Haskell with futures.
In *Proceedings of the 13th international ACM SIGPLAN symposium on Principles and practices of declarative programming*, PPDP '11, S. 101–112. ACM, New York, NY, USA, 2011.

DOI: 10.1145/2003476.2003492

9. D. Sabel
An abstract machine for Concurrent Haskell with futures.
In *Software Engineering 2012 - Workshopband, Fachtagung des GI-Fachbereichs Softwaretechnik, 27. Februar - 2. März 2012 in Berlin (5. Arbeitstagung Programmiersprache, ATPS 2012)*, Bd. 199 von *GI Edition - Lecture Notes in Informatics*, S. 29–44. Köllen Druck+Verlag, Bonn, Germany, 2012.

URL: http://subs.emis.de/LNI/Proceedings/Proceedings199/article6673.html

10. D. Sabel & M. Schmidt-Schauß
Conservative concurrency in Haskell.
In *Proceedings of the 27th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2012), 25-28 June 2012, Dubrovnik, Croatia*, S. 561–570, IEEE, 2012.

DOI: 10.1109/LICS.2012.66

11. C. Rau, D. Sabel & M. Schmidt-Schauß
Correctness of program transformations as a termination problem.
In *Automated Reasoning – Proceedings of the 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012*, Bd. 7364 von *Lecture Notes in Computer Science*, S. 462–476. Springer Berlin / Heidelberg, 2012.

DOI: 10.1007/978-3-642-31365-3_36

12. D. Sabel, & M. Schmidt-Schauß
A two-valued logic for properties of strict functional programs allowing partial functions.
Journal of Automated Reasoning, Springer, *Journal of Automated Reasoning*, 50(4):383–421, 2013.

DOI: 10.1007/s10817-012-9253-6

Zu vielen der Arbeiten gibt es technische Berichte (insbesondere (Schmidt-Schauß et al., 2008a; Schmidt-Schauß & Sabel, 2008; Schmidt-Schauß et al., 2009b; Schwinghammer et al., 2009a; Schmidt-Schauß et al., 2009a; Sabel & Schmidt-Schauß, 2011d; Schmidt-Schauß et al., 2010a; Sabel & Schmidt-Schauß, 2011a,c; Sabel, 2012a)), die genauere Details und aufwändigere Beweise enthalten. Die technischen Berichte sind allesamt unter der URL *http://www.ki.informatik.uni-frankfurt.de/papers/frank/* abrufbar.

In Kapitel 2 werden die Grundlagen der folgenden Kapitel erläutert. Die Notation des Programmkalküls wird als grundlegendes Modell für Programmiersprachen definiert und anhand einfacher Beispiele illustriert. Im Anschluss werden grundlegende Notationen und Begriffe erläutert. Hierbei wird insbesondere der Begriff der kontextuellen Gleichheit motiviert, definiert und von anderen Gleichheitsbegriffen abgegrenzt. In diesem Kapitel werden in Abschnitt 2.3 bereits die Resultate aus (Schmidt-Schauß & Sabel, 2010a) (Veröffentlichung (6) in obiger Liste) zu Abschlusseigenschaften drei verschiedener Konvergenzbegriffe bezüglich der kontextuellen Gleichheit erörtert. Das Kapitel schließt mit einem Überblick über verschiedene Korrektheitsbegriffe für die verschiedenen Anwendungsbereiche und bereitet daher die Klassifizierung der Ergebnisse nachfolgender Kapitel vor.

In Kapitel 3 werden die Veröffentlichungen (1) und (2) aus obiger Liste zusammengefasst. Dabei werden zunächst die Ergebnisse zur Untersuchung von Korrektheitsbegriffen für Übersetzungen zwischen Programmkalkülen und für Implementierungen von Programmkonstrukten aus (Schmidt-Schauß et al., 2008b) dargestellt. Im Anschluss werden die Ergebnisse aus (Schwinghammer et al., 2009b) erläutert, welche als größeres Anwendungsbeispiel die Korrektheit der Implementierung verschiedener Nebenläufigkeitsprimitive im Rahmen einer strikten, impuren, nebenläufigen Programmiersprache untersucht.

In Kapitel 4 werden Aspekte zum syntaktischen Nachweis von kontextueller Gleichheit und der Korrektheit von Programmtransformationen erörtert, indem die Arbeiten (5) und (11) aus obiger Liste zusammengefasst werden. Dabei wird zuerst über die Gültigkeit eines sogenannten Kontextlemmas für eine Reihe von Programmkalkülen berichtet, welche in (Schmidt-Schauß & Sabel, 2010b) nachgewiesen wurde. Im Anschluss werden die Ergebnisse aus (Rau et al., 2012) zur Automatisierung der Induktion innerhalb der sogenannten Diagrammmethode dargestellt. Diese Methode stellt ein syntaktisches Verfahren zum Korrektheitsnachweis von Programmtransformationen dar.

In Kapitel 5 werden die Ergebnisse der drei Arbeiten (3), (4) und (7) dargestellt. Alle drei Arbeiten befassen sich mit call-by-need Lambda-Kalkülen mit einem `letrec`-Konstrukt, die als Kernsprachen für verzögert auswertende funktionale Programmiersprache, wie z. B. Has-

kell, verwendet werden. Ein besonderes Augenmerk richtet sich dabei auf den Nachweis der Korrektheit von Programmtransformationen mithilfe der sogenannten Bisimulationsgleichheit, die als Hilfsmittel dient, um kontextuelle Gleichheit zu folgern. Für den reinen call-by-need Lambda-Kalkül mit rekursiven let-Ausdrücken zeigt (Schmidt-Schauß et al., 2010b), dass Bisimulation und kontextuelle Gleichheit zusammenfallen und letztendlich sogar die Isomorphie dieses Kalküls mit dem Lazy-Lambda-Kalkül, der kein letrec-Konstrukt besitzt und eine call-by-name Auswertung verwendet. Ein negatives Resultat wird in (Schmidt-Schauß et al., 2011) nachgewiesen, indem gezeigt wird, dass für nichtdeterministische Kalküle mit rekursiven let-Ausdrücken und Datenkonstruktoren die Bisimulationsgleichheit nicht verträglich mit der kontextuellen Äquivalenz ist. Schließlich wird über die Ergebnisse aus (Sabel et al., 2009) berichtet, welche die Korrektheit von Programmtransformation unter polymorpher Typisierung untersucht.

Kapitel 6 stellt die Untersuchungen zum sogenannten CHF-Kalkül aus den Arbeiten (8), (9) und (10) der obigen Liste dar. Der CHF-Kalkül modelliert Concurrent Haskell mit Futures, die Erweiterung der funktionalen Programmiersprache Haskell um nebenläufige Threads, synchronisierende Speicherzellen und nebenläufige Futures. Zunächst werden die Untersuchungen aus (Sabel & Schmidt-Schauß, 2011b) dargestellt, die den CHF-Kalkül einschließlich einer kontextuellen Semantik definieren, eine Reihe von Programmtransformationen als korrekt zeigen und die Äquivalenz der call-by-name und call-by-need Auswertungsstrategie beweisen. Die Analyse aus (Sabel & Schmidt-Schauß, 2012) zeigt, dass der CHF-Kalkül eine konservative Erweiterung der rein funktionalen Kernsprache von Haskell ist und daher bekannte Methoden und Gleichheiten auf den CHF-Kalkül übertragen werden können. Schließlich werden die Untersuchungen zu einem korrekten abstrakten Maschinenmodell für den CHF-Kalkül aus (Sabel, 2012b) erläutert.

In Kapitel 7 werden die Resultate aus (Sabel & Schmidt-Schauß, 2013) (Publikation (12) aus obiger Liste) erörtert. In (Sabel & Schmidt-Schauß, 2013) wird die Programmlogik LMF für eine strikte funktionale Programmiersprache auf Basis der kontextuellen Äquivalenz eingeführt. Wesentliche Eigenschaften der Logik werden erläutert, wobei ein besonderes Augenmerk auf die Konservativität der lokalen Gültigkeit logischer Formeln hin zur globalen Gültigkeit gerichtet ist. Des Weiteren werden Eigenschaften bestehender Logiken erörtert und mit der Logik LMF verglichen.

In Kapitel 8 schließen wir diese zusammenfassende Darstellung mit einem kurzen Fazit.

2
Grundlagen

In diesem Kapitel werden die grundlegenden Eigenschaften von Programmkalkülen, der Gleichheitsbegriff von Programmen und Fragestellungen zur Korrektheit von Programmen und Programmtransformationen erörtert.

In Abschnitt 2.1 wird ein einheitliches Modell für Programmiersprachen als Programmkalkül eingeführt und es werden einfache Beispiele für solche Kalküle gegeben. Der Begriff der kontextuellen Äquivalenz wird in Abschnitt 2.2 für deterministische und nichtdeterministische Programmkalküle eingeführt, motiviert und von anderen Gleichheitsbegriffen abgegrenzt. Da kontextuelle Äquivalenz auf der Beobachtung der Konvergenz aufbaut, werden in Abschnitt 2.3 die Ergebnisse aus (Schmidt-Schauß & Sabel, 2010a) erörtert, die Abschlusseigenschaften von drei Konvergenzprädikaten (May-, Must- und Should-Konvergenz) nachweisen und die Verwendung der Should-Konvergenz anstelle der Must-Konvergenz motivieren. Schließlich wird in Abschnitt 2.4 die Notwendigkeit und die Abgrenzung von Korrektheitsbegriffen in verschiedenen Bereichen der Informatik erläutert, um die in den anschließenden Kapiteln vorgestellten Arbeiten zu klassifizieren und zu motivieren.

2.1 Programmkalküle

Abstrakt fassen wir eine Programmiersprache als einen Programmkalkül auf, den wir an dieser Stelle allgemein – also als Rahmen, passend für viele Programmiersprachen – definieren.

Definition 2.1.1 (Getypter Programmkalkül)**.** *Ein* getypter Programmkalkül *ist ein 5-Tupel* $D = (\mathcal{T}, \mathcal{E}, \mathcal{C}, \xrightarrow{sr}, \mathcal{A})$ *wobei:*

- $\mathcal{T}$ *ist die Menge der* Typen *von* D*.*

- *$\mathcal{E}$ ist die Menge der getypten* Ausdrücke *des Programmkalküls. Jeder Ausdruck $e \in \mathcal{E}$ hat einen Typ $T \in \mathcal{T}$. Wir schreiben $\mathcal{E}_T$ für die Menge der Ausdrücke vom Typ T.*

- *$\mathcal{C}$ ist die Menge der* Kontexte *von D, wobei jeder Kontext $C \in \mathcal{C}$ eine Funktion ist, die getypte Ausdrücke auf getypte Ausdrücke abbildet (d. h. $C : \mathcal{E}_T \to \mathcal{E}_{T'}$). Wir schreiben $\mathcal{C}_{T,T'}$ für diejenigen Kontexte, die Ausdrücke vom T auf Ausdrücke vom Typ T' abbilden. Wir nehmen dabei an, dass für jeden Typ T stets die Identitätsfunktion (der leere Kontext) in der Menge $\mathcal{C}$ enthalten ist, und dass $\mathcal{C}$ bezüglich der Komposition von Kontexten abgeschlossen ist, d. h. für alle $C_1 \in \mathcal{C}_{T_1,T_3}$ und $C_2 \in \mathcal{C}_{T_2,T_1}$ gilt $(C_1 \circ C_2) \in \mathcal{C}_{T_2,T_3}$.*

- *$\xrightarrow{sr} \subseteq (\mathcal{E} \times \mathcal{E})$ ist die* Standardreduktion, *die Ausdrücke in andere Ausdrücke überführt. Wenn $e_1 \xrightarrow{sr} e_2$, dann sagen wir e_1* reduziert zu *e_2. Gibt es für einen Ausdruck $e_1 \in \mathcal{E}$ keinen Ausdruck e_2, sodass $e_1 \xrightarrow{sr} e_2$, so nennen wir e_1* irreduzibel *(und anderenfalls* reduzibel*). Wir nehmen an, dass die Standardreduktion* typerhaltend *ist, d. h. wenn $e_1 \xrightarrow{sr} e_2$ gilt, dann sind e_1 und e_2 stets vom selben Typ.*

- *$\mathcal{A} \subseteq \mathcal{E}$ ist die Menge der* Antworten *(oft auch Werte oder Normalformen genannt).*

Ein Programmkalkül ist deterministisch, *wenn aus $e_1 \xrightarrow{sr} e_2$ und $e_1 \xrightarrow{sr} e_3$ stets $e_2 = e_3$ folgt. Hierbei meint $=$ die syntaktische Gleichheit von Ausdrücken, die oft so definiert ist, dass sie bis auf Umbenennung gleiche Ausdrücke ebenfalls identifiziert (man spricht in diesem Fall von α-äquivalenten Ausdrücken).*

Üblicherweise sind die Antworten $\mathcal{A}$ irreduzibel bezüglich $\xrightarrow{sr}$, wir setzen dies jedoch an dieser Stelle nicht voraus.

Die Standardreduktion ist üblicherweise als *Small-Step*-Reduktion definiert, d. h. ein Schritt $e_1 \xrightarrow{sr} e_2$ führt eine kleine Modifikation des Ausdrucks e_1 hin zu Ausdruck e_2 durch und das Erhalten einer Antwort $a \in \mathcal{A}$ wird erreicht, indem mehrere Standardreduktionen nacheinander ausgeführt werden. Wir verwenden die Notationen $\xrightarrow{sr,n}$ für n Reduktionsschritte, $\xrightarrow{sr,+}$ für einen oder mehre (aber endlich viele) Reduktionsschritte, $\xrightarrow{sr,*}$ für 0 oder mehr Reduktionsschritte und $\xrightarrow{sr,\omega}$ für unendlich viele Reduktionsschritte. Im Gegensatz zu *Small-Step*-Reduktionen werden des Öfteren auch *Big-Step*-Reduktionen verwendet. Dabei führt ein Schritt stets direkt zu einer Antwort. Eine Big-Step-Reduktion lässt oft Freiheiten in der genauen Implementierung eines Interpreters, während eine Small-Step-Reduktion die *operationale Semantik* der Programmiersprache genauer beschreibt und die direkte Ableitung eines Interpreters ermöglicht. Wir nehmen im Folgenden an, dass $\xrightarrow{sr}$ eine Small-Step-Reduktion ist.

Die Menge der Kontexte wird meist direkt aus der Menge der Ausdrücke konstruiert, indem jeder Unterausdruck jedes Ausdrucks durch das Kontextloch (notiert mit $[\cdot]$) ersetzt wird. Die Anwendung eines Kontextes C (als Funktion) auf einen Ausdruck e entspricht in diesem Fall der *Einsetzung* von e in das Loch des Kontextes C, notiert als $C[e]$, dabei wird $[\cdot]$ in C durch e (ohne Umbenennung!) ersetzt.

Ein ungetypter Programmkalkül besitzt keine Typen, allerdings kann dessen Definition auf die Definition des getypten Kalküls zurückgeführt werden, indem ein einziger Typ Ausdruck

hinzugefügt wird, sodass jeder Ausdruck genau diesen Typ besitzt. Daher ist die Notation des getypten Programmkalküls das allgemeinere Konzept.

Definition 2.1.2 (Ungetypter Programmkalkül)**.** *Ein* ungetypter Programmkalkül *ist ein 4-Tupel* $D = (\mathcal{E}, \mathcal{C}, \xrightarrow{sr}, \mathcal{A})$*, sodass* $D' = (\{\mathsf{Ausdruck}\}, \mathcal{E}, \mathcal{C}, \xrightarrow{sr}, \mathcal{A})$ *ein getypter Programmkalkül gemäß Definition 2.1.1 ist, wobei jeder Ausdruck* $e \in \mathcal{E}$ *vom Typ* Ausdruck *ist, und jeder Kontext* $C \in \mathcal{C}$ *eine Funktion* $C : \mathsf{Ausdruck} \to \mathsf{Ausdruck}$ *ist.*

2.1.1 Einfache Beispiele

Wir betrachten einige Varianten des Lambda-Kalküls (der ursprünglich von Church eingeführt wurde (Church, 1941), ein Standardwerk zum Lambda-Kalkül ist (Barendregt, 1984)) als Beispiele für Programmkalküle.

Beispiel 2.1.3. *Der Lambda-Kalkül mit Normalordnungsreduktion (Barendregt, 1984) kann als als ungetypter Programmkalkül aufgefasst werden. Sei* $L_{no} := (\mathcal{E}_{no}, \mathcal{C}_{no}, \xrightarrow{no}, \mathcal{A}_{no})$ *der Lambda-Kalkül mit Normalordnungsreduktion, wobei die einzelnen Komponenten wiefolgt festgelegt sind: Ausdrücke* $\mathcal{E}_{no}$ *sind durch die Grammatik*

$$e, e_i \in \mathcal{E}_{no} ::= x \mid \lambda x.e \mid (e_1\ e_2)$$

festgelegt. Ausdrücke bestehen daher aus Variablen x *aus einer unendlichen Menge von Variablen, aus* Abstraktionen $\lambda x.e$*, und* Anwendungen $(e_1\ e_2)$ *In* $\lambda x.e$ *ist die Variable* x *mit* Geltungsbereich e gebunden*. Variablen, die nicht gebunden sind, werden* freie Variablen *genannt. Mit* $FV(e)$ *bezeichnen wir die Menge der freien Variablen des Ausdrucks* e*. Wenn* $FV(e) = \emptyset$*, dann sagen wir* e *ist* geschlossen*. Für sämtliche Kalküle verwenden wir Barendregts Variablenkonvention (Barendregt, 1984), und nehmen an, dass die Namen freier Variablen paarweise verschieden von gebundenen Variablen sind und dass die Namen gebundener Variablen paarweise verschieden sind. Durch* α*-Umbenennungen (entsprechend der Regel* $\lambda x.e \to \lambda y.e[y/x]$*, wobei* y *ein neuer Name ist und* $e[y/x]$ *die Ersetzung aller freien Vorkommen von* x *in* e *durch* y *ist) kann diese Konvention stets eingehalten werden. Wir nehmen auch an, dass die Standardreduktion solche Umbenennungen durchführt, um die Konvention stets einzuhalten.*

Kontexte $\mathcal{C}_{no}$ *sind alle Ausdrücke, die an einer Stelle ein Loch* $[\cdot]$ *besitzen.*

Eine β-Reduktion *ist die Ersetzungsregel* $(\lambda x.e_1)\ e_2 \xrightarrow{\beta} e_1[e_2/x]$ *(wobei* $e_1[e_2/x]$ *bedeutet, dass alle freien Vorkommen der Variablen* x *im Ausdruck* e_1 *durch den Ausdruck* e_2 *ersetzt werden). Die Standardreduktion* $\xrightarrow{no}$ *wird als Normalordnungsreduktion bezeichnet und wendet die* β*-Reduktion soweit links und soweit außen wir möglich an (sie wird daher auch als leftmost-outermost Strategie bezeichnet). Die Normalordungsreduktion ist eindeutig (bis auf* α*-Umbenennungen), daher ist* L_{no} *ein deterministischer Kalkül. Antworten* $\mathcal{A}_{no}$ *sind alle Terme die in Normalform bezüglich der Normalordungsreduktion sind, d .h. die keinen* β*-Redex*[1] *besitzen.*

Der Lambda-Kalkül mit Normalordungsreduktion eignet sich im Allgemeinen nicht als Modell für funktionale Programmiersprachen, da diese nicht in den Rümpfen von Abstraktionen

[1] Redex = reducible expression

Ausdrücke: $e, e_i \in \mathcal{E}_{lazy} ::= x \mid \lambda x.e \mid (e_1\ e_2)$
Kontexte: $C \in \mathcal{C}_{lazy} ::= [\cdot] \mid \lambda x.C \mid (C\ e) \mid (e\ C)$
Antworten: $a \in \mathcal{A}_{lazy} ::= \lambda x.e$
Standardreduktion: $R[((\lambda x.e_1)\ e_2)] \xrightarrow{lazy} R[e_1[e_2/x]]$ wobei $R \in \mathcal{R}_{lazy} ::= [\cdot] \mid (R\ e)$

Abbildung 2.1: Der Kalkül L_{lazy}

auswerten, sondern die Auswertung als erfolgreich beenden, sobald eine Abstraktion (eine sogenannte schwache Kopfnormalform) erhalten wurde. Diese Auswertung wird durch den Lazy-Lambda-Kalkül modelliert:

Beispiel 2.1.4. *Abramskys Lazy-Lambda-Kalkül (Abramsky, 1990) kann als ungetypter Programmkalkül in obiger Notation aufgefasst werden. Sei $L_{lazy} := (\mathcal{E}_{lazy}, \mathcal{C}_{lazy}, \xrightarrow{lazy}, \mathcal{A}_{lazy})$ der Lazy-Lambda-Kalkül, wobei die einzelnen Komponenten in Abbildung 2.1 festgelegt sind. Ausdrücke $\mathcal{E}_{lazy}$ bzw. Kontexte $\mathcal{C}_{no}$ entsprechen dabei Ausdrücken bzw. Kontexten des Lambda-Kalküls mit Normalordnungsreduktion.*

Die Standardreduktion $\xrightarrow{lazy}$ wendet die β-Reduktion in Reduktionskontexten *$\mathcal{R}_{lazy}$ an. Da die Standardreduktion $\xrightarrow{lazy}$ eindeutig (bis auf α-Umbenennungen) ist, ist der Lazy-Lambda-Kalkül ein deterministischer Programmkalkül.*

Die Antworten $\mathcal{A}_{lazy}$ sind alle Abstraktionen $\lambda x.e$, die in L_{lazy} auch schwache Kopfnormalformen (WHNF) genannt werden.

Als weiteres Beispiel betrachten wir erneut den Lazy-Lambda-Kalkül, wobei wir jedoch anstelle eines Ersetzungssystems eine abstrakte Maschine als operationale Semantik verwenden. Dieses Beispiel demonstriert, dass auch abstrakte Maschinen in den Rahmen eines Programmkalküls passen:

Beispiel 2.1.5. *Eine einfache abstrakte Maschine für den Lazy-Lambda-Kalkül kann als ungetypter Programmkalkül wie folgt dargestellt werden: Die Menge der Ausdrücke besteht aus Zuständen der Form (e, S), wobei $e \in \mathcal{E}_{lazy}$ ein Ausdruck des Lazy-Lambda-Kalküls ist (siehe Beispiel 2.1.4) und S ein Stack von L_{lazy}-Ausdrücken ist. Wir schreiben $[]$ für den leeren Stack und $e : S$ für den Stack, der e als oberstes Element enthält.*

Kontexte für die abstrakte Maschine sind alle Funktionen f mit $f((e, S)) = (C[e], S)$, wobei $C \in \mathcal{C}_{lazy}$ ein Kontext des Lazy-Lambda-Kalkül ist (siehe Beispiel 2.1.4). Antworten sind alle Zustände der Form $(\lambda x.e, [])$. Die Standardreduktion ist durch die folgenden beiden Maschinentransitionen definiert:

$$\begin{array}{lcl} (\lambda x.e_1, e_2 : S) & \to & (e_1[e_2/x], S) \\ ((e_1\ e_2), S) & \to & (e_1, e_2 : S) \end{array}$$

Die erste Transition entspricht dabei der β-Reduktion und die zweite Transition entspricht der Suche nach dem nächsten Redex.

Ausdrücke: $e, e_i \in \mathcal{E}_{value} ::= x \mid \lambda x.e \mid (e_1\ e_2)$
Kontexte: $C \in \mathcal{C}_{value} ::= [\cdot] \mid \lambda x.C \mid (C\ e) \mid (e\ C)$
Antworten: $a \in \mathcal{A}_{value} ::= x \mid \lambda x.e$
Standardreduktion: $R[((\lambda x.e_1)\ a)] \xrightarrow{value} R[e_1[a/x]]$
wobei $R \in \mathcal{R}_{value} ::= [\cdot] \mid (R\ e) \mid (a\ R)$ und $a \in \mathcal{A}_{value}$

Abbildung 2.2: Der Kalkül L_{value}

Während der Lazy-Lambda-Kalkül die call-by-name Auswertungsstrategie verfolgt, also Funktionseinsetzung ohne vorherige Auswertung der Argumente durchführt, wertet der call-by-value Lambda-Kalkül (siehe z. B. (Plotkin, 1975)) zunächst das Argument einer Anwendung aus, bevor es in den Rumpf der Abstraktion eingesetzt werden darf.

Beispiel 2.1.6. *Der ungetypte Programmkalkül $L_{value} := (\mathcal{E}_{value}, \mathcal{C}_{value}, \xrightarrow{value}, \mathcal{A}_{value})$ ist der* call-by-value Lambda-Kalkül*, wobei die Komponenten in Abbildung 2.2 definiert sind. Die Ausdrücke entsprechen den Ausdrücken des Lazy-Lambda-Kalküls. Antworten umfassen neben Abstraktionen auch Variablen. Die Standardreduktion wendet eine eingeschränkte (β)-Reduktion (das Argument muss eine Antwort sein) in call-by-value Reduktionskontexten an. Die Reduktion wertet daher erst das Argument einer Anwendung aus, bevor die (β)-Reduktion angewendet werden kann. Diese Reduktion ist eindeutig (bis auf α-Umbenennungen) und daher ist L_{value} ein deterministischer Programmkalkül.*

Als Beispiel für einen getypten Programmkalkül betrachten wir eine Variante des einfach getypten Lambda-Kalküls (Church, 1940).

Beispiel 2.1.7. *Der einfach getypte call-by-name Lambda-Kalkül mit Fixpunkt-Operator ist ein getypter Programmkalkül gemäß Definition 2.1.1 definiert durch $L^{\tau}_{lazyf} = (\mathcal{T}^{\tau}_{lazyf}, \mathcal{E}^{\tau}_{lazyf}, \mathcal{C}^{\tau}_{lazyf}, \xrightarrow{lazyf}, \mathcal{A}^{\tau}_{lazyf})$. Die Typen $\mathcal{T}^{\tau}_{lazyf}$ des Kalküls sind durch die Grammatik $T \in \mathcal{T}^{\tau}_{lazyf} ::= o \mid T \to T$ festgelegt. Dabei ist o der* Basistyp *und Typen der Form $T \to T$ sind* Funktionstypen.

Die Grammatik $e \in \mathcal{E}_{lazyf} ::= x \mid \lambda x.e \mid (e_1\ e_2) \mid$ fix legt zunächst die ungetypten Ausdrücke fest. Neben den Konstrukten des Lazy-Lambda-Kalküls wurde die Konstante fix hinzugefügt, da ansonsten sämtliche Ausdrücke des einfach getypten Lambda-Kalküls terminierend sind. Für die Menge der getypten Ausdrücke $\mathcal{E}^{\tau}_{lazyf}$ wird ein Typsystem benutzt, welches zum einen nicht wohl-getypte Ausdrücke aus $\mathcal{E}_{lazyf}$ aussortiert und zum anderen den Typ jedes wohl-getypten Ausdrucks ermittelt. Der Einfachheit halber nehmen wir an, dass Variablen bereits mit einem (eingebauten) Typ $T \in \mathcal{T}^{\tau}_{lazyf}$ versehen sind, wir schreiben daher x^T, wenn die Variable x den eingebauten Typ T besitzt. Die Typisierungsregeln sind als logische Schlussregeln zu lesen, wobei wir die Bruchstrichnotation verwenden: Im Zähler steht die Voraussetzung und im Nenner die Konsequenz.

$$\frac{e_1 :: T_1 \to T_2, e_2 :: T_2}{(e_1\ e_2) :: T_2} \qquad \frac{}{x^T :: T} \qquad \frac{e :: T_2}{(\lambda x^{T_1}.e) :: T_1 \to T_2} \qquad \frac{}{\mathit{fix} :: (T \to T) \to T}$$

Ein Ausdruck e ist genau dann wohl-getypt mit Typ T, wenn $e :: T$ mit obigen Regeln herleitbar ist.

Die Grammatik $C \in \mathcal{C}_{lazyf} ::= [\cdot] \mid (C\ e) \mid (e\ C) \mid \lambda x.C$ legt die Syntax von ungetypten Kontexten $\mathcal{C}_{lazyf}$ fest. Getypte Kontexte $C \in \mathcal{C}^{\tau}_{lazyf}$ werden ebenfalls durch obiges Typsystem bestimmt: Ein Kontext C ist wohl-getypt, wenn C wie ein Ausdruck getypt werden kann, wobei für das Kontextloch das Axiom

$$\frac{}{[\cdot] :: T}$$

für alle Typen T verwendet werden darf. Wenn T am Loch verwendet wurde und der Kontext anschließend mit dem Typ T' typisiert wurde, dann gehört der Kontext zur Menge $\mathcal{C}^{\tau}_{lazyf,T,T'}$, d. h. man kann Ausdrücke vom Typ T in den Kontext einsetzen und erhält als Resultat einen Ausdruck vom Typ T'.

Die Standardreduktion $\xrightarrow{lazyf}$ wendet β-Reduktion oder fix-Reduktion in einem Reduktionskontext an, wobei Reduktionskontexte und β-Reduktion wie im Lazy-Lambda-Kalkül definiert sind und die fix-Reduktion durch die Regel fix $e \xrightarrow{fix} e$ (fix e) definiert ist. Man kann leicht nachweisen, dass die Reduktion typerhaltend ist.

Antworten $\mathcal{A}^{\tau}_{lazyf}$ sind alle wohl-getypten Abstraktionen sowie die Konstante fix.

Schließlich betrachten wir als letztes Beispiel einen nichtdeterministischen Programmkalkül, der in ähnlicher Variante beispielsweise in (Ong, 1993) zu finden ist.

Beispiel 2.1.8. *Der call-by-name Lambda-Kalkül mit erratischer Auswahl ist ein ungetypter Programmkalkül $L_{lazynd} = (\mathcal{E}_{lazynd}, \mathcal{C}_{lazynd}, \xrightarrow{lazynd}, \mathcal{A}_{lazynd})$, wobei die Komponenten in Abbildung 2.3 festgelegt sind. Ausdrücke erweitern den Lazy-Lambda-Kalkül um sogenannte* choice-Ausdrücke *$e_1 \oplus e_2$. Die Standardreduktion führt β- und choice-Reduktionen in Reduktionskontexten durch. Die choice-Reduktionen sind*

$$\begin{array}{ll} \textit{(choicel)} & e_1 \oplus e_2 \to e_1 \\ \textit{(choicer)} & e_1 \oplus e_2 \to e_2 \end{array}$$

Sie implementieren gerade den erratischen Nichtdeterminismus. Der call-by-name Lambda-Kalkül mit erratischer Auswahl ist daher ein nichtdeterministischer Programmkalkül, da beispielsweise $(\lambda x.\lambda y.x) \oplus (\lambda x.\lambda y.y)$ sowohl zu $\lambda x.\lambda y.x$ als auch zu $\lambda x.\lambda y.y$ reduzieren kann.

Für weitere Beispiele definieren wir die folgenden Abkürzungen für spezifische Ausdrücke des Lambda-Kalküls:

$$\begin{array}{rcl} \mathbf{\Omega} & := & (\lambda x.x\ x)\ (\lambda x.x\ x) \\ \mathbf{I} & := & \lambda x.x \\ \mathbf{Y} & := & \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)) \end{array}$$

Beachte, dass $\mathbf{\Omega}$ und $\mathbf{Y}$ nicht typisierbar sind, und daher keine gültigen Ausdrücke in λ^{τ}_{lazyf} darstellen. In den ungetypten Kalkülen besitzt $\mathbf{\Omega}$ die Eigenschaft, auf sich selbst zu reduzieren, d. h. $\mathbf{\Omega} \xrightarrow{lazy} \mathbf{\Omega}$ und daher auch $\mathbf{\Omega} \xrightarrow{lazy,\omega}$. $\mathbf{I}$ ist die Identitätsfunktion. $\mathbf{Y}$ ist ein sogenannter Fixpunktoperator. Für ihn gilt $\mathbf{Y}\ e \xrightarrow{lazy} \mathbf{Y}[e/f] \xrightarrow{lazy,*} e\ (Y[e/f])$.

Ausdrücke: $e, e_i \in \mathcal{E}_{lazynd} ::= x \mid \lambda x.e \mid (e_1\ e_2) \mid (e_1 \oplus e_2)$
Kontexte: $C \in \mathcal{C}_{lazynd} ::= [\cdot] \mid \lambda x.C \mid (C\ e) \mid (e\ C) \mid (C \oplus e) \mid (e \oplus C)$
Antworten: $a \in \mathcal{A}_{lazynd} ::= \lambda x.e$
Standardreduktion: (mit $R \in \mathcal{R}_{lazynd} ::= [\cdot] \mid (R\ e)$)

$$R[((\lambda x.e_1)\ e_2)] \xrightarrow{lazynd} R[e_1[e_2/x]]$$
$$R[e_1 \oplus e_2] \xrightarrow{lazynd} R[e_1]$$
$$R[e_1 \oplus e_2] \xrightarrow{lazynd} R[e_2]$$

Abbildung 2.3: Der Kalkül L_{lazynd}

2.2 Gleichheit

Ein wesentlicher Begriff im Rahmen der Korrektheit von Programmen und Programmtransformationen ist der Begriff der *Gleichheit* von Ausdrücken. Er ist notwendig, um gleiche Programme zu identifizieren und verschiedene Programme voneinander zu unterscheiden. Betrachten wir beispielsweise eine Programmoptimierung, die durch den Compiler einer Programmiersprache durchgeführt wird (z. B. das Entfernen von unbenutztem Code, das Entfalten von Prozedurrümpfen, etc.). Mithilfe eines Gleichheitsbegriffes für Ausdrücke können wir festlegen, ob diese Optimierung korrekt oder fehlerhaft ist, denn es sollten nur Unterprogramme durch *gleiche* Unterprogramme ersetzt werden.

Damit möglichst viele Optimierungen als korrekt nachgewiesen werden können, sollte der Gleichheitsbegriff möglichst weitläufig sein, und daher nur wirklich verschiedene Programme voneinander unterscheiden. Eine weitere Forderung an die Gleichheit ist, dass deren Definition möglichst *kanonisch* ist, d. h. sie sollte möglichst für viele Programmiersprachen in der gleichen Form definierbar sein. Dies verhindert, dass für jede neue Programmiersprache ein neuer (möglicherweise völlig verschiedener) Begriff entwickelt und gefunden werden muss. Eine der wesentlichsten Forderungen an die Gleichheit ist daher ebenso, dass sie allgemein als richtig akzeptiert ist. Aus mathematischer aber auch aus praktischer Sicht ist es zudem notwendig, dass der Gleichheitsbegriff eine *Kongruenz* ist, d. h. die Gleichheit sollte eine Äquivalenzrelation auf Ausdrücken bilden, welche kompatibel mit der Einsetzung in Kontexte ist: Die Forderung nach einer Äquivalenzrelation ergibt sich daraus, dass identische Ausdrücke stets als gleich anzusehen sind (Reflexivität); wenn e_1 gleich zu e_2 ist, dann sollte auch e_2 gleich zu e_1 sein (Symmetrie); und wenn e_1 gleich zu e_2 und e_2 gleich zu e_3 ist, dann sollten auch e_1 und e_3 als gleich identifiziert werden (Transitivität). Kompatibilität mit Kontexten erfordert, dass die Gleichheit stabil bezüglich der Einsetzung in Kontexte ist, d. h. wenn e_1 und e_2 gleiche Ausdrücke sind, dann sollten auch $C[e_1]$ und $C[e_2]$ für beliebe Kontexte C als gleich identifiziert werden. Diese Forderung kann vor allem dadurch begründet werden, dass man Unterprogramme korrekt transformieren möchte *ohne* das äußere Programm (den Kontext) zu beachten. Beispielsweise will man $1 + 1$ stets durch 2 ersetzen, unabhängig davon, in welchem

Programm dieser arithmetische Ausdruck steht. Ebenso spielt diese Eigenschaft eine wichtige Rolle bei separater Compilierung von Modulen: Die Korrektheit von verwendeten Optimierungen bei der Compilation eines Moduls soll nicht von Änderungen an anderen Modulen abhängen.

2.2.1 Konvertibilität

Oft wird der Begriff der Konvertibilität als Gleichheitsbegriff verwendet, der besagt, dass zwei Programme gleich sind, wenn sie mithilfe der Kalkülreduktionen (üblicherweise die Regeln der Standardreduktion, allerdings ausgedehnt auf alle Kontexte) ineinander *überführbar* sind. Für den Lazy-Lambda-Kalkül kann dieser Begriff definiert werden als: $e_1 \overset{*}{\leftrightarrow} e_2$ genau dann, wenn e_1 durch Anwenden beliebiger Beta-Reduktionen (in beiden Richtungen) in e_2 überführt werden kann. Allerdings ist dieser Begriff der Gleichheit im Allgemeinen zu schwach, um alle objektiv gleichen Programme miteinander zu identifizieren, denn die Programme müssen in die syntaktisch gleiche Form gebracht werden. Betrachte hierzu beispielsweise zwei verschiedene Sortierverfahren auf Zahlen, wie Mergesort und Quicksort. Obwohl beide Verfahren das Gleiche leisten (wenn auch mit unterschiedlichen Laufzeiten), kann Konvertibilität im Allgemeinen deren Gleichheit nicht zeigen. Konvertibilität könnte für dieses Beispiel gezeigt werden, wenn man die Sortierverfahren auf spezifischen Eingaben betrachtet, da die Ausgaben in diesem Fall Datenwerte sind. Wenn man jedoch Funktionen höherer Ordnung betrachtet, die als Ausgaben Funktionen liefern können, wird dieser Ansatz zumindest komplizierter (man müsste das Verfahren iterativ anwenden).

Gänzlich falsch wird der Begriff der Konvertibilität, wenn man einen nichtdeterministischen Programmkalkül betrachtet, da offensichtlich unterschiedliche Programme durch den Konvertibilitätsbegriff identifizierbar sind. Beispielsweise sind nichtdeterministische Programme konvertibel mit deterministischen Programmen. Z. B. sind die beiden Ausdrücke $\mathbf{I} \oplus \mathbf{\Omega}$ und $\mathbf{I}$ konvertibel im call-by-name Lambda Kalkül mit erratischer Auswahl, wobei jedoch der erste Ausdruck in einem niemals terminierenden Programm enden kann, während der zweite Ausdruck ein Wert (die Identitätsfunktion) ist.

Ein Problem der Verwendung des Konvertibilitätsbegriffs als Gleichheitsnotation liegt darin, dass a priori angenommen wird, dass (Standard-)reduktionen gleichheitserhaltend sind, was jedoch zumindest bei nichtdeterministischen Kalkülen (oder z. B. auch Kalkülen mit Seiteneffekten) nicht immer der Fall ist.

2.2.2 Kontextuelle Gleichheit

Einen passenden Gleichheitsbegriff liefert die kontextuelle Gleichheit, die im Allgemeinen verschieden von der Konvertibilität ist. Die kontextuelle Gleichheit identifiziert Ausdrücke nur dann als gleich, wenn der Austausch des einen Ausdrucks durch einen anderen im beliebigen Kontext nicht beobachtbar ist. Formaler definiert sind zwei Ausdrücke e_1 und e_2 kontextuell gleich, wenn sie sich in allen Kontexten gleich verhalten (d. h. $C[e_1]$ und $C[e_2]$ verhalten sich gleich). Dies spezifiziert die Gleichheit noch nicht vollends, da das beobachtete Verhalten

noch nicht definiert ist. Eine Möglichkeit ist es, die erhaltenen Antworten nach erschöpfendem Reduzieren mit der Standardreduktion als Beobachtung zu verwenden. Allerdings hat dieser Ansatz einerseits Schwächen, denn Funktionen höherer Ordnung können Funktionen als Ergebnis liefern, die jedoch nicht syntaktisch als gleiche Werte identifizierbar sind, und zum anderen führt der Test in allen Kontexten dazu, dass es oft nicht notwendig ist, gleiche Werte zu beobachten, da unterschiedliche Werte im Allgemeinen durch (kompliziertere) Kontexte unterschieden werden können. Daher reicht es aus, die Möglichkeiten der *Terminierung* zu betrachten.

Wir definieren die sogenannte May-Konvergenz allgemein für Programmkalküle:

Definition 2.2.1 (May-Konvergenz)**.** *Sei $D = (\mathcal{T}, \mathcal{E}, \mathcal{C}, \xrightarrow{sr}, \mathcal{A})$ ein getypter Programmkalkül. Ein Ausdruck $e \in \mathcal{E}$* may-konvergiert*, geschrieben als $e{\downarrow_D}$, genau dann, wenn e auf eine Antwort mithilfe endlich vieler Standardreduktionen reduziert werden kann, d. h.*

$$e{\downarrow_D} \text{ gdw. } \exists e' \in \mathcal{A} : e \xrightarrow{sr,*} e'$$

Wenn $e{\downarrow_D}$ nicht gilt, dann sagen wir e ist must-divergent *und schreiben dies als $e{\Uparrow_D}$*

In einem deterministischen Programmkalkül reicht die Betrachtung der May-Konvergenz für die Definition der kontextuellen Gleichheit aus. Der Begriff der kontextuellen Äquivalenz geht dabei auf (Morris, 1968) zurück.

Definition 2.2.2 (Kontextuelle Präordnung und Äquivalenz bzgl. $\downarrow$)**.** *Sei $D = (\mathcal{T}, \mathcal{E}, \mathcal{C}, \xrightarrow{sr}, \mathcal{A})$ ein getypter Programmkalkül. Dann sind die* kontextuelle Präordnung $\leq_\downarrow$ *und die* kontextuelle Äquivalenz $\sim_\downarrow$ *definiert als:*

- *Für Ausdrücke $e_1, e_2 \in \mathcal{E}_T$ gilt $e_1 \leq_\downarrow e_2$ gdw. $\forall T' \in \mathcal{T}, C \in \mathcal{C}_{T,T'} : C[e_1]{\downarrow_D} \implies C[e_2]{\downarrow_D}$.*
- *Für Ausdrücke $e_1, e_2 \in \mathcal{E}_T$ gilt $e_1 \sim_\downarrow e_2$ gdw. $e_1 \leq_D e_2$ und $e_2 \leq_D e_1$.*

Damit diese Ordnungs- und Gleichheitsbegriffe stets offensichtlich unterschiedliche Programme unterscheiden, sollte die Menge $\mathcal{C}$ der Kontexte, alle denkbaren Programme (Ausdrücke) mit Loch umfassen. Die Definition der kontextuellen Gleichheit zeigt, dass der Begriff kanonisch ist, denn er kann für alle deterministischen Programmkalkül definiert werden, ohne genauere Interna des spezifischen Kalküls zu kennen.

Allgemein gilt, dass sowohl kontextuelle Gleichheit als auch kontextuelle Ungleichheit unentscheidbar sind, da sie einfach auf das Halteproblem zurück geführt werden können. Beispielsweise ist der Lazy-Lambda-Kalkül Turing-mächtig, und daher entspricht die Frage, ob $e \not\sim_{lazy} \Omega$ gilt, gerade dem Halteproblem. Der Definition der kontextuellen Gleichheit ist jedoch zu entnehmen, dass es oft vergleichsweise einfach ist, die *Ungleichheit* zweier Ausdrücke nachzuweisen, da lediglich ein Kontext gefunden werden muss, der die beiden Ausdrücke bezüglich der May-Konvergenz unterscheidet.

Beispiel 2.2.3. *Im Lazy-Lambda-Kalkül sind die Ausdrücke $\lambda x.x$ und $\lambda x.(x\ x)$ kontextuell verschieden, da der Kontext $C := ([\cdot]\ \lambda y.(y\ y))$ die beiden Ausdrücke unterscheidet: Während $C[\lambda x.x] \xrightarrow{lazy} \lambda y.(y\ y)$ und daher may-konvergiert, gilt $C[\lambda x.(x\ x)] \xrightarrow{lazy} C[\lambda x.(x\ x)] \xrightarrow{lazy} C[\lambda x.(x\ x)] \xrightarrow{lazy} \ldots$, d. h. $C[\lambda x.(x\ x)]$ ist must-divergent.*

Der Nachweis der kontextuellen Gleichheit zweier Ausdrücke ist im Gegensatz zur Widerlegung wesentlich schwieriger, da Konvergenzübereinstimmung für die im Allgemeinen abzählbar unendliche Menge von Kontexten nachgewiesen werden muss. Daher werden im Allgemeinen Hilfsmittel und Beweistechniken benötigt, um solche Nachweise zu erbringen. In Kapitel 4 werden einige solcher Hilfsmittel erörtert.

Für nichtdeterministische Programmkalküle ist die alleinige Betrachtung der May-Konvergenz in der Definition der kontextuellen Gleichheit im Allgemeinen nicht ausreichend, da (analog zur Argumentation in Abschnitt 2.2.1 für die Konvertibilität) konvergierende Ausdrücke mit manchmal divergierenden und manchmal konvergierenden Ausdrücken gleichgesetzt werden. Z. B. unterscheidet die kontextuelle Gleichheit aus Definition 2.2.2 die beiden Ausdrücke des Lazy-Lambda-Kalküls mit erratischer Auswahl $\mathbf{I} \oplus \mathbf{\Omega}$ und $\mathbf{I}$ nicht, obwohl $\mathbf{I}$ stets konvergiert, $\mathbf{I} \oplus \mathbf{\Omega}$ jedoch auch zu dem must-divergenten Ausdruck $\mathbf{\Omega}$ auswerten kann. Daher erfordert die kontextuelle Gleichheit für nichtdeterministische Kalküle die Betrachtung, ob ein Ausdruck stets terminiert oder nicht. Hierbei unterscheidet die Literatur zwei verschiedene Begriffe, die wir zum einen mit Should-Konvergenz (siehe z. B. (Carayol et al., 2005; Niehren et al., 2007; Sabel & Schmidt-Schauß, 2008; Sabel, 2008)) und zum anderen mit Must-Konvergenz (siehe z. B. (de'Liguoro & Piperno, 1992; Alessi et al., 1994; Moran et al., 1999; Levy, 2007)) bezeichnen[2]. Die Definitionen sind:

Definition 2.2.4 (Should- und Must-Konvergenz)**.** *Sei $D = (\mathcal{T}, \mathcal{E}, \mathcal{C}, \xrightarrow{sr}, \mathcal{A})$ ein getypter Programmkalkül. Ein Ausdruck $e \in \mathcal{E}$* should-konvergiert, *geschrieben als $e{\Downarrow_D}$, genau dann, wenn alle Nachfolger von e bezüglich der Standardreduktion may-konvergent sind, d. h.*

$$e{\Downarrow_D} \textit{ gdw. } \forall e' \in \mathcal{E} : e \xrightarrow{sr,*} e' \implies e'{\downarrow_D}.$$

Ist ein Ausdruck e nicht should-konvergent, so sagen wir e may-divergiert und schreiben dies als $e{\uparrow_D}$.

Ein Ausdruck $e \in \mathcal{E}$ must-konvergiert, *geschrieben als $e{\Downarrow\!\Downarrow_D}$, genau dann, wenn e should-konvergiert und es keine unendlich lange Folge von Standardreduktionen beginnend mit e gibt, d. h.*

$$e{\Downarrow\!\Downarrow_D} \textit{ gdw. } e{\Downarrow} \wedge \neg(e \xrightarrow{sr,\omega})$$

Alternativ lässt sich die Must-Konvergenz dadurch charakterisieren, dass jede mögliche Auswertung (Reduktionsfolge) nach endlich vielen Schritten mit einer Antwort endet. Should- und Must-Konvergenz sind im Allgemeinen unterschiedliche Prädikate für Ausdrücke, wobei Must-Konvergenz offensichtlich Should-Konvergenz impliziert. Beide Prädikate erlauben die Definition weiterer kontextueller Präordnungen und die Kombination dieser Präordnungen führt zu verschiedenen Begriffen der kontextuellen Gleichheit

Für die Definition der kontextuellen Präordnung und Gleichheit wird in der Literatur meist entweder die Kombination aus May- und Must-Konvergenz, oder die Kombination aus May- und Should-Konvergenz verwendet.

[2]Teilweise werden in der Literatur andere Begriffe verwendet, z. B. verwendet (Schmidt-Schauß & Sabel, 2010b) den Begriff Must-Konvergenz anstelle von Should-Konvergenz, und totale Must-Konvergenz anstelle von Must-Konvergenz. Wir verwenden hier ausschließlich die Begriffe Should- und Must-Konvergenz im Sinne von Definition 2.2.4.

Definition 2.2.5 (Kontextuelle Präordnungen und Äquivalenzen für Should- und Must-Konvergenz)**.** *Sei $D = (\mathcal{T}, \mathcal{E}, \mathcal{C}, \xrightarrow{sr}, \mathcal{A})$ ein getypter Programmkalkül. Dann sind die* kontextuelle Präordnungen $\leq_{\downarrow}$ *und* $\leq_{\Downarrow}$ *definiert als*

- *Für Ausdrücke $e_1, e_2 \in \mathcal{E}_T$ gilt $e_1 \leq_{\downarrow} e_2$ gdw. $\forall T' \in \mathcal{T}, C \in \mathcal{C}_{T,T'} : (C[e_1]{\downarrow}_D \implies C[e_2]{\downarrow}_D)$.*
- *Für Ausdrücke $e_1, e_2 \in \mathcal{E}_T$ gilt $e_1 \leq_{\Downarrow} e_2$ gdw. $\forall T' \in \mathcal{T}, C \in \mathcal{C}_{T,T'} : (C[e_1]{\Downarrow}_D \implies C[e_2]{\Downarrow}_D)$.*

Entsprechend definieren wir die Schnitte von verschiedenen kontextuellen Präordnungen und deren zugehörige kontextuelle Äquivalenzen als:

$$\begin{array}{lcl} \leq_{\downarrow,\Downarrow} & := & \leq_{\downarrow} \cap \leq_{\Downarrow} \\ \leq_{\downarrow,\Downarrow} & := & \leq_{\downarrow} \cap \leq_{\Downarrow} \\ \sim_{\downarrow,\Downarrow} & := & \leq_{\downarrow,\Downarrow} \cap \geq_{\downarrow,\Downarrow} \\ \sim_{\downarrow,\Downarrow} & := & \leq_{\downarrow,\Downarrow} \cap \geq_{\downarrow,\Downarrow} \end{array}$$

Wendet man diese Definitionen auf einen deterministischen Kalkül an, so erhält man $\sim_{\downarrow} = \sim_{\downarrow,\Downarrow} = \sim_{\downarrow,\Downarrow}$, da in diesem Fall May-, Should-, und Must-Konvergenz jeweils das identische Prädikat auf Ausdrücken darstellen.

Für nichtdeterministische Kalküle sind im Allgemeinen alle drei Relationen verschieden:

Beispiel 2.2.6 (Carayol et al. (2005))**.** *Betrachten wir die Ausdrücke $e_1 := \mathbf{I}$, $e_2 := \mathbf{Y}\ \lambda f.(\mathbf{I} \oplus f)$ und $e_3 := (\mathbf{I} \oplus \mathbf{\Omega})$ im Lazy-Lambda-Kalkül mit erratischer Auswahl L_{lazynd}. e_1 ist die Identitätsfunktion, e_3 enthält einen choice-Operator, dessen Auswertung darüber entscheidet, ob e_3 zur Identitätsfunktion reduziert oder zum divergierenden Ausdruck $\mathbf{\Omega}$. e_2 lässt sich entfalten als $\mathbf{I} \oplus (\mathbf{I} \oplus (\mathbf{I} \oplus \ldots))$, d. h. entweder die Auswertung endet mit $\mathbf{I}$, oder die Auswertung endet nicht, indem ständig das rechte Argument jedes choice-Ausdrucks gewählt wird. Trotz allem verbleibt jeder Nachfolger bezüglich der Standardreduktion stets may-konvergent. Man kann nachweisen, dass im Lazy-Lambda-Kalkül mit erratischer Auswahl gilt: $e_1 \sim_{\downarrow} e_2 \sim_{\downarrow} e_3$, aber $e_1 \not\sim_{\downarrow,\Downarrow} e_2 \sim_{\downarrow,\Downarrow} e_3$ und $e_1 \sim_{\downarrow,\Downarrow} e_2 \not\sim_{\downarrow,\Downarrow} e_3$.*

Aus den oben genannten Gründen ist die alleinige Betrachtung der May-Konvergenz in diesem Fall nicht ausreichend (e_1 und e_3 aus dem vorangegangenen Beispiel sollten als verschieden klassifiziert werden). Es sollte daher eine der Varianten $\sim_{\downarrow,\Downarrow}$ bzw. $\sim_{\downarrow,\Downarrow}$ als Gleichheitsbegriff verwendet werden. Wir bevorzugen erstere Variante, da diese Kombination einige vorteilhafte Eigenschaften aufweist. Ein Vorteil liegt darin, dass im Allgemeinen Should-Konvergenz unverändert bleibt, wenn die Auswertungsstrategie auf faire Reduktionsfolgen eingeschränkt wird, was bei nebenläufigen Programmiersprachen eine sinnvolle Annahme ist. Daher kann zum Gleichheitsnachweis eine operationale Semantik verwendet werden, die nicht auf Fairness achtet, aber sämtliche Gleichheitsresultate sind auf die Semantik mit Fairnessbachtung übertragbar, da die induzierten kontextuellen Gleichheiten genau zusammenfallen. Dies erleichtert Beweise erheblich, da operationale Semantiken, die Fairness beachten sehr aufwändig und kompliziert sind. Must-Konvergenz hat diese Eigenschaft im Allgemeinen nicht, d. h. die induzierten kontextuellen Gleichheiten bezüglich Must-Konvergenz sind echt verschieden, jenachdem, ob Fairness beachtet wird oder nicht. Da die faire Auswertung der sinnvolle Begriff ist (gerade im Hinblick auf sogenannte Busy-Wait-Implementierungen in nebenläufigen

Programmiersprachen), erfordert die Verwendung der Must-Konvergenz daher das explizite Betrachten der operationalen Semantik mit Fairnesseinschränkungen.

Diskussionen und entsprechende Resultate hierzu sind beispielsweise in (Carayol et al., 2005; Rensink & Vogler, 2007; Niehren et al., 2007; Sabel, 2008; Sabel & Schmidt-Schauß, 2008, 2011a) zu finden.

Einen weiteren Vorteil der Should-Konvergenz im Gegensatz zur Must-Konvergenz werden wir im Abschnitt 2.3 erläutern.

2.2.3 Applikative Bisimulation als alternativer Gleichheitsbegriff

Applikative Bisimulation (siehe z. B. (Howe, 1989, 1996; Abramsky, 1990; Gordon, 1994, 1999)) ist ein alternativer Gleichheitsbegriff, der zwei Ausdrücke als gleich identifiziert, wenn sie durch Reduktion zu Werten, anschließendem Anwenden der Werte auf Argumente und iterieren dieses Verfahrens nicht unterschieden werden können. Je nach Programmkalkül kann dieses Anwenden auf Argumente auch beispielsweise das Extrahieren von Untertermen beinhalten (z. B. bei komplexen Datenwerten). Bei nichtdeterministischen Programmkalkülen sind zudem die erhaltenen Werte nicht eindeutig und daher müssen alle Möglichkeiten in Betracht gezogen werden. Wir geben in diesem Abschnitt keine allgemeine Definition der Bisimulation an, da sie mehr Information über die Syntax des Programmkalküls benötigt, als sie durch Definition 2.1.1 gegeben ist. Außerdem beschränken wir uns auf den Test der May-Konvergenz.

Im Lazy-Lambda-Kalkül kann Bisimulation wie folgt definiert werden:

Definition 2.2.7 (Bisimulation in L_{lazy})**.** *Sei F_{lazy} der folgende Operator auf binären Relationen über geschlossenen Ausdrücken des Lazy-Lambda-Kalküls: Sei η eine binäre Relation über geschlossenen Ausdrücken, dann gilt $(e_1, e_2) \in F_{lazy}(\eta)$ genau dann, wenn: Wenn $e_1 \xrightarrow{lazy,*} \lambda x.e_1'$, dann $e_2 \xrightarrow{lazy,*} \lambda x.e_2'$ und für alle geschlossenen Ausdrücke r gilt $(((\lambda x.e_1')\ r), ((\lambda x.e_1')\ r)) \in \eta$.*

Applikative Simulation $\leq_{b,lazy}$ *ist definiert als der größte Fixpunkt von F_{lazy}.* Applikative Bisimulation $\sim_{b,lazy}$ *ist definiert als $\sim_b := \leq_b \cap \geq_b$. Die* offene Erweiterung $\sim^o_{b,lazy}$ *von $\sim_{b,lazy}$ als Relationen auf allen (auch offenen) Ausdrücken des Lazy-Lambda-Kalküls ist definiert als $e_1 \sim^o_{b,lazy} e_2$ genau dann, wenn $\sigma(e_1) \sim_{b,lazy} \sigma(e_2)$ für alle Substitutionen σ, die Variablen durch geschlossene Ausdrücke ersetzen und für die $FV(\sigma(e_1)) \cup FV(\sigma(e_2)) = \emptyset$ gilt.*

Applikative Bisimulation ist daher coinduktiv definiert und verlangt im Allgemeinen coinduktive Beweise[3]. Der Vorteil gegenüber der Definition der kontextuellen Äquivalenz liegt darin, dass Kontexte im Gleichheitsbeweis nicht betrachtet werden müssen.

Allerdings ist applikative Bisimulation im Allgemeinen nur dann nützlich, wenn sie zumindest korrekt bezüglich der kontextuellen Äquivalenz ist, d. h. aus $e_1 \sim_b e_2$ folgt auch die kontextuelle Gleichheit von e_1 und e_2. Im Lazy-Lambda-Kalkül gilt sogar Vollständigkeit, d. h. $\sim^o_{b,lazy} = \sim_\downarrow$ (siehe (Abramsky, 1990)). Für eine Reihe von deterministischen und auch nichtdeterministischen – vornehmlich call-by-name auswertenden – Programmkalkülen kann diese Vollständigkeit mithilfe der Methode von Howe (Howe, 1989, 1996; Pitts, 2011) nachgewiesen

[3] Eine gelungene Einführung zur Coinduktion ist (Gordon, 1994).

werden. Bei nichtdeterministischen Kalkülen muss die Definition der applikativen Simulation angepasst werden, wir führen dies beispielhaft für den Kalkül L_{lazynd} vor:

Definition 2.2.8 (Bisimulation in L_{lazynd})**.** *Sei F_{lazynd} der folgende Operator auf binären Relationen über geschlossenen Ausdrücken von L_{lazynd}. Sei η eine binäre Relation über geschlossenen Ausdrücken, dann gilt $(e_1, e_2) \in F_{lazynd}(\eta)$ genau dann, wenn: Für alle Abstraktionen $\lambda x.e_1'$ mit $e_1 \xrightarrow{lazynd,*} \lambda x.e_1'$, existiert eine Abstraktion $\lambda x.e_2'$ mit $e_2 \xrightarrow{lazynd,*} \lambda x.e_2'$, sodass für alle geschlossenen Ausdrücke r gilt $(((\lambda x.e_1')\ r), ((\lambda x.e_1')\ r)) \in \eta$.*

Applikative Simulation $\leq_{b,lazynd}$ *ist definiert als der größte Fixpunkt von F_{lazynd}.* Applikative Bisimulation $\sim_{b,lazynd}$ *ist definiert als $\sim_b := \leq_b \cap \geq_b$. Die* offene Erweiterung $\sim^o_{lazynd}$ *ist analog wie in L_{lazy} definiert.*

Howes Methode ist auch für L_{lazynd} anwendbar. Die Methode findet jedoch ihre Grenze bei syntaktisch komplexeren Programmkalkülen, da sie einige syntaktische Bedingungen an den Programmkalkül beinhaltet. Insbesondere müssen die verschiedenen syntaktischen Konstrukte in sogenannte kanonische und nichtkanonische Konstrukte getrennt werden, wobei ein Ausdruck genau dann eine Antwort ist, wenn der oberste Operator kanonisch ist (Howe, 1989). Für den Lazy-Lambda-Kalkül ist beispielsweise der Operator λ kanonisch und die Applikation nicht kanonisch. In komplexeren Programmkalkülen gibt es jedoch Programmkonstrukte, die diese Einteilung nicht erfüllen (z. B. die Konstrukte `let` und `letrec`). In diesen Fällen ist es erheblich schwieriger eine korrekte Definition der applikativen Bisimulation zu finden (einige Fälle werden z. B. in den Arbeiten (Lassen, 1998; Lassen & Pitcher, 2000; Mann, 2005b,a; Mann & Schmidt-Schauß, 2010) behandelt) und deren Korrektheit oder gar Vollständigkeit zu beweisen (ein aufwändiger Beweis wird in Abschnitt 5.1 erläutert). Manchmal ist es sogar unmöglich eine korrekte applikative Bisimulation zu definieren (siehe Abschnitt 5.2).

2.3 Abschlusseigenschaften von May-, Should- und Must-Konvergenz

In Abschnitt 2.2.2 wurden May-, Should- und Must-Konvergenz eingeführt und die entsprechenden kontextuellen Gleichheiten bezüglich dieser Beobachtungsprädikate definiert. Die Definition der Should-Konvergenz (Definition 2.2.4) erinnert dabei an den Notwendigkeitsoperator der Modallogik (Hughes & Cresswell, 1990). Daher wurde in (Schmidt-Schauß & Sabel, 2010a) untersucht, ob durch modallogische Konstruktionen weitere Beobachtungsprädikate konstruiert werden können, die möglicherweise zu Verfeinerungen der Definition der entsprechenden kontextuellen Äquivalenz führen. In diesem Abschnitt werden wir die Resultate aus (Schmidt-Schauß & Sabel, 2010a) zusammenfassen.

Hierfür wurden zunächst allgemein für (ungetypte) Programmkalküle die folgenden Prädikatgeneratoren definiert, welche die Verknüpfung bestehender Prädikate mithilfe aussagenlogischer und den modallogischen Operatoren $\Box$ und $\Diamond$ erlauben.

Definition 2.3.1. *Sei $D = (\mathcal{E}, \mathcal{C}, \xrightarrow{st}, \mathcal{A})$ ein ungetypter Programmkalkül und seien $P, Q \subseteq \mathcal{E}$ Prädikate auf Ausdrücken, dann seien die Prädikate $\Box P, \Diamond P, \neg P, P \vee Q$ und $P \wedge Q$ definiert als:*

$$\begin{aligned}
\Box P &:= \{e \in \mathcal{E} \mid \forall e' : e \xrightarrow{sr,*} e' \implies P(e')\} \\
\Diamond P &:= \{e \in \mathcal{E} \mid \exists e' : e \xrightarrow{sr,*} e' \wedge P(e')\} \\
P \vee Q &:= P \cup Q \\
P \wedge Q &:= P \cap Q \\
\neg P &:= \mathcal{E} \setminus P
\end{aligned}$$

Für eine Menge von Prädikaten $\mathcal{P}$ (mit $P \subseteq \mathcal{E}$ für alle $P \in \mathcal{P}$) sei $B^{\Box}_{\Diamond}(\mathcal{P})$ der Abschluss von $\mathcal{P}$ bezüglich aller obigen Operationen, und $N^{\Box}_{\Diamond}(\mathcal{P})$ der Abschluss von $\mathcal{P}$ bezüglich $\Box, \Diamond$ und $\neg$. Die kontextuelle Präordnung bezüglich $\mathcal{P} = \{P_1, \ldots, P_n\}$ (geschrieben als $\leq_{\mathcal{P}}$) ist definiert als $\leq_{\mathcal{P}} := \bigcap_{i=1}^{n} \leq_{P_i}$ mit $e_1 \leq_{P_i} e_2$ gdw. $\forall C \in \mathcal{C} : P_i(C[e_1]) \implies P_i(C[e_2])$. Die kontextuelle Äquivalenz bezüglich $\mathcal{P}$ ist definiert als $e_1 \sim_{\mathcal{P}} e_2$ gdw. $e_1 \leq_{\mathcal{P}} e_2$ und $e_2 \leq_{\mathcal{P}} e_1$.

Zum Beispiel entspricht $\Diamond\mathcal{A}$ gerade der May-Konvergenz $\downarrow$ und $\Box\downarrow = \Box\Diamond\mathcal{A}$ entspricht der Should-Konvergenz $\Downarrow$.

Das wesentlichste Resultat aus (Schmidt-Schauß & Sabel, 2010a) besagt, dass die kontextuelle Äquivalenz bezüglich $\{B^{\Box}_{\Diamond}\downarrow\}$ genau der kontextuellen Äquivalenz bezüglich $\{\downarrow, \Downarrow\}$ entspricht, d. h. Ausdrücke e_1, e_2 mit $e_1 \sim_{\downarrow,\Downarrow} e_2$ können auch nicht bezüglich aller Prädikate unterschieden werden (in allen Kontexten), die durch obige Operatoren aus der May-Konvergenz gebildet werden können. Wir wiederholen dieses Resultat mitsamt einfacher Beschreibungen für $N^{\Box}_{\Diamond}(\{\downarrow\})$ und $B^{\Box}_{\Diamond}(\{\downarrow\})$:

Theorem 2.3.2. *Für jeden Programmkalkül gilt:*

- $N^{\Box}_{\Diamond}(\{\downarrow\}) = \{\downarrow, \uparrow, \Downarrow, \Uparrow\}$
- $B^{\Box}_{\Diamond}(\{\downarrow\}) = \{\emptyset, \downarrow, \uparrow, \Downarrow, \Uparrow, \downarrow \wedge \uparrow, \Downarrow \vee \Uparrow, \mathcal{E}\}$
- $\sim_{\downarrow,\Downarrow} = \sim_{B^{\Box}_{\Diamond}\{\downarrow\}}$

Hinsichtlich der Must-Konvergenz konnte in (Schmidt-Schauß & Sabel, 2010a) gezeigt werden, dass sie diese Abschlusseigenschaft *nicht* besitzt:

Satz 2.3.3. *Im Allgemeinen gilt, dass die Menge $B^{\Box}_{\Diamond}(\{\Uparrow\})$ unendlich viele Prädikate enthält. Daher gibt es i. A. keine endliche Menge von Prädikaten $\mathcal{P} \subseteq B^{\Box}_{\Diamond}(\{\Uparrow\})$ mit $\sim_{\mathcal{P}} = \sim_{B^{\Box}_{\Diamond}(\{\Uparrow\})}$.*

Zusammenfassend zeigen die Ergebnisse, dass die kontextuelle Äquivalenz bezüglich May- und Should-Konvergenz die Abschlusseigenschaft besitzt und daher eine ganze Klasse von Beobachtungsprädikaten umfasst. Die analoge Argumentation ist für die kontextuelle Äquivalenz basierend auf May- und Must-Konvergenz nicht möglich: Die durch den Abschluss bezüglich dieser Prädikate definierte kontextuelle Gleichheit erfordert es, das gleiche Verhalten von Ausdrücken bezüglich unendlich vieler Beobachtungsprädikate nachzuweisen. Negativ aus Sicht der Must-Konvergenz ausgedrückt, kann man argumentieren, dass die Wahl der Beobachtungsprädikate May- und Must-Konvergenz in der Definition der kontextuellen Äquivalenz wahllos erscheint, da beliebig viele weitere Beobachtungsprädikate sinnvoll hinzugefügt werden könnten.

2.4 Korrektheitsbegriffe und Fragestellungen

Im Rahmen der Korrektheit von Programmen ergeben sich vielfältige Fragestellungen, die wir an dieser Stelle kurz diskutieren werden, um spätere Kapitel entsprechend den Fragestellungen zuzuordnen.

Aus der Definition der kontextuellen Gleichheit ergibt sich zunächst die Frage nach Anwendungsbereichen für welche ein Nachweis der kontextuellen Gleichheit von Ausdrücken notwendig, sinnvoll oder hilfreich ist. Ein Anwendungsgebiet ist die Untersuchung der Korrektheit von *Programmtransformationen*.

Definition 2.4.1. *Sei* $D = (\mathcal{T}, \mathcal{E}, \mathcal{C}, \xrightarrow{sr}, \mathcal{A})$ *ein getypter Programmkalkül und* $\sim_D$ *die zugehörige kontextuelle Äquivalenz. Eine* Programmtransformation *ist eine binäre Relation* $\xrightarrow{T}$ *auf Ausdrücken* $\mathcal{E}$*, wobei für alle* $e_1 \xrightarrow{T} e_2$ *die Ausdrücke* e_1, e_2 *stets vom gleichen Typ sind (d. h.* $e_1, e_2 \in \mathcal{E}_T$ *für irgendeinen Typ* $T \in \mathcal{T}$*). Eine Programmtransformation* $\xrightarrow{T}$ *ist* korrekt*, wenn sie die kontextuelle Gleichheit erhält, d. h. es gilt*

$$\xrightarrow{T} \text{ ist korrekt gdw. } \xrightarrow{T} \subseteq \sim_D$$

Um die Korrektheit einer Programmtransformation zu zeigen, ist daher der Beweis der kontextuellen Gleichheit für alle in der Transformationen enthaltenen Ausdruckspaaren zu erbringen. Programmtransformationen selbst kommen in verschiedenen Bereichen der Programmiersprachen und Programmierung vor: In *Compilern* werden Programmtransformationen verwendet, um Programme zu *optimieren*. Dabei stellt die Korrektheit der Transformationen sicher, dass der Compiler die Semantik des Programms erhält. Prominente Beispiele sind das Entfernen von nicht verwendetem Code (häufig „Garbage Collection“ genannt), das Entfalten von Prozedurrümpfen (sogenanntes „Procedure Inlining“), das partielle Auswerten von Unterausdrücken, etc. (z. B. sind eine Reihe solcher Transformationen in (Peyton Jones & Santos, 1994) im Compiler GHC[4] für die funktionale Programmiersprache Haskell[5] (Peyton Jones, 2003) beschrieben).

Im Rahmen des *„Program Refactoring“* werden Programmtransformationen verwendet, um die Lesbarkeit des Quellcodes zu verbessern. Hierbei ist es unerlässlich, dass die Transformationen semantikerhaltend und daher korrekte Programmtransformationen sind.

Werkzeuge zur *Software-Verifikation* benötigen Programmtransformationen, um Aussagen über Programme zu beweisen, die notwendigerweise korrekt sein müssen.

Durch diese Vielfältigkeit der Anwendungen von Programmtransformationen ist es unerlässlich, Korrektheit von Programmtransformationen tatsächlich in konkreten Programmkalkülen nachzuweisen. Während der Lambda-Kalkül und ähnlich syntaktisch kleine Programmkalküle hier gut untersucht sind, sind Kalküle, die realistische Programmiersprachen und insbesondere moderne Programmiersprachen mit Nebenläufigkeit modellieren, eher weniger untersucht. In Kapitel 6 werden wir die Untersuchung eines solchen Kalküls zur Modellierung

[4] *http://haskell.org/ghc*
[5] *http://haskell.org*

von Concurrent Haskell erörtern, die insbesondere den Korrektheitsnachweis von Programmtransformationen für diesen syntaktisch reichen Kalkül umfasst. In Abschnitt 5.3 wird zudem erläutert, wie Korrektheitsresultate aus einem ungetypten Programmkalkül für den funktionalen Kern der Programmiersprache Haskell auf den polymorph getypten Fall übertragen werden können. Aber auch in anderen Arbeiten spielen korrekte Programmtransformationen eine wichtige Rolle, da sie notwendig sind, um weitere Eigenschaften der entsprechenden Programmkalküle zu beweisen.

Es ist zudem notwendig, Werkzeuge und Hilfsmittel zu entwickeln, die Korrektheitsbeweise von Programmtransformationen erleichtern. In Kapitel 4 werden einige solcher Werkzeuge präsentiert und zudem Untersuchungen zu deren Automatisierbarkeit erörtert.

Ein anderes Anwendungsgebiet im Rahmen der Korrektheit von Programmen ist die Untersuchung der Korrektheit von *Übersetzungen* zwischen Programmkalkülen. Solche Übersetzungen treten beispielsweise im Compiler auf, wenn meist mehrstufige Übersetzungen die Quellsprache in die Zielsprache übersetzen. Diese Übersetzungen können jedoch auch dazu dienen, die *Ausdruckskraft* von Programmkalkülen zu vergleichen. Für die Korrektheit von Übersetzungen ist der Nachweis von kontextuellen Gleichheiten zwar oft ein nützliches Werkzeug, aber im Vordergrund steht der Vergleich der kontextuellen Äquivalenz zwischen zwei (möglicherweise völlig verschiedenen Sprachen). Die entsprechenden Korrektheitsbegriffe, Anwendungsbereiche und Anwendungsbeispiele werden wir in Kapitel 3 genauer diskutieren. In Kapitel 5 werden diese Begriffe und Notationen schließlich im Rahmen einer Untersuchung zur Ausdruckskraft des `letrec`-Konstrukts und dem Lazy-Lambda-Kalkül verwendet. Dadurch wird der Beweis der Korrektheit der applikativen Bisimulation im um `letrec` erweiterten Lazy-Lambda-Kalkül ermöglicht.

Schließlich gibt es im Rahmen der Softwareverifikation und entsprechend in automatischen Theorembeweisern weitere Korrektheitsbegriffe, die sich nicht ausschließlich auf die Gleichheit von Programmen beziehen, sondern im Allgemeinen die Gültigkeit von *Aussagen über Programmen* meint. Genauer werden dabei Aussagen über Programme innerhalb einer Logik formuliert, um im Anschluss die Tautologieeigenschaft, die Erfüllbarkeit oder auch die Widersprüchlichkeit entsprechender logischer Formeln nachzuweisen. Ein einfaches Beispiel ist die Aussage, dass für alle natürlichen Zahlen $x, y > 0$ gilt: $x + y \neq 0$. Im Kern dieser Aussage ist jedoch auch eine Ungleichheit versteckt. In Kapitel 7 werden wir einen Ansatz erläutern, wie eine solche Logik auf Basis der kontextuellen Gleichheit aufgebaut werden kann.

3

Korrektheit von Übersetzungen und Implementierungen

In diesem Kapitel betrachten wir zum einen die Korrektheit von Übersetzungen zwischen Programmkalkülen, wie sie beispielsweise bei der Compilierung von Programmiersprachen auftreten, wenn syntaktisch reichere Kalküle zunächst in Kernsprachen übersetzt und schließlich in Maschinen- oder Maschinennahen Code übersetzt werden. Zum anderen betrachten wir auch die Korrektheit von Implementierungen neuer Programmkonstrukte, wie sie beispielsweise in Form von Programmbibliotheken in vielen Programmiersprachen zu finden sind. Dieses Kapitel fasst dabei die Ergebnisse der Veröffentlichungen (Schmidt-Schauß et al., 2008b) und (Schwinghammer et al., 2009b) zusammen.

In (Schmidt-Schauß et al., 2008b) wurden die grundlegenden Begriffe und Eigenschaften von Übersetzungen erarbeitet, wobei sich der Begriff der Beobachtungskorrektheit als wesentlicher Korrektheitsbegriff herausgestellt hat. Die erarbeiteten Eigenschaften und Notation finden sich auch in vielen weiteren Arbeiten wieder, die in späteren Kapiteln erörtert werden, und daher auch Anwendungsfälle des allgemeinen Rahmens aus (Schmidt-Schauß et al., 2008b) darstellen.

In (Schwinghammer et al., 2009b) werden diese Eigenschaften angewendet, um die Korrektheit verschiedener Implementierungen von Primitiven der nebenläufigen Programmierung in einer Kernsprache der nebenläufigen, strikten, funktionalen Programmiersprache Alice ML nachzuweisen.

3.1 Übersetzungen zwischen Programmkalkülen

In (Schmidt-Schauß et al., 2008b) wurden allgemein Eigenschaften von Übersetzungen zwischen Programmkalkülen und deren kontextuellen Gleichheiten untersucht. Programmkalküle werden dabei in einem allgemeinem Rahmen ähnlich wie in Definition 2.1.1 repräsentiert,

wobei anstelle der Antworten $\mathcal{A}$ und der Reduktionsrelation $\xrightarrow{D}$, eine Menge $\mathcal{O}$ von Beobachtungsprädikaten (wie beispielsweise May-Konvergenz, Must-Konvergenz, etc.) in der Kalkülbeschreibung gegeben ist. D. h. ein getypter Programmkalkül wird als 4-Tupel $D = (\mathcal{T}, \mathcal{E}, \mathcal{C}, \mathcal{O})$ dargestellt, wobei $\mathcal{T}$ die Menge der Typen, $\mathcal{E}$ die Menge der Ausdrücke, $\mathcal{C}$ die Menge der Kontexte und $\mathcal{O} = \{\Downarrow_1, \ldots, \Downarrow_n\}$ eine Menge von Beobachtungsprädikaten auf Ausdrücken ist. Die kontextuelle Präordnung $\leq_D$ ist definiert als $\bigcap_{i=1}^{n} \leq_{\Downarrow_i}$, wobei $\leq_{\Downarrow_i}$ gerade die kontextuelle Präordnung bzgl. des Beobachtungsprädikats $\Downarrow_i$ ist (d. h. für $e_1, e_2 \in \mathcal{E}_T$ gilt $e_1 \leq_{\Downarrow_i} e_2$ genau dann, wenn $\forall T' \in \mathcal{T}, C \in \mathcal{C}_{T,T'} : C[e_1]\Downarrow_i \implies C[e_2]\Downarrow_i$).

Definition 3.1.1. *Eine* Übersetzung $\sigma : D \to D'$ *vom Kalkül* $D = (\mathcal{T}, \mathcal{E}, \mathcal{C}, \{\Downarrow_1, \ldots, \Downarrow_n\})$ *in den Kalkül* $D' = (\mathcal{T}', \mathcal{E}', \mathcal{C}', \{\Downarrow'_1, \ldots, \Downarrow'_n\})$*, bildet Typen auf Typen, Ausdrücke auf Ausdrücke, Kontexte auf Kontexte und* $\Downarrow_i$ *auf* $\Downarrow'_i$ *ab, wobei gelten muss*

- *wenn* $\sigma(e) = e'$ *und* $e \in \mathcal{E}_T$*, dann gilt* $e' \in \mathcal{E}'_{\sigma(T)}$
- *wenn* $\sigma(C) = C'$ *und* $C \in \mathcal{C}_{T_1,T_2}$*, dann gilt* $C' \in \mathcal{C}'_{\sigma(T_1),\sigma(T_2)}$
- $\sigma([\cdot]) = [\cdot]$*, d. h. der leere Kontext (oder entsprechend – die Identitätsfunktion) wird auf den leeren Kontext (bzw. die Identitätsfunktion) abgebildet.*

Es werden daher nur Übersetzungen zwischen Kalkülen betrachtet, die über die gleiche Anzahl an Beobachtungsprädikaten verfügen und die Prädikate werden eins zu eins durch die Übersetzung abgebildet.

Insbesondere wurde in (Schmidt-Schauß et al., 2008b) erörtert, welche Anforderungen an eine Übersetzung zu stellen sind, damit diese als „korrekt“ bezeichnet werden kann. Je nach Art der Anwendung kann man hier unterschiedliche Korrektheitsbegriffe unterscheiden, wir erörtern einige Anwendungsbereiche für Übersetzungen zwischen Programmkalkülen:

Ein Compiler übersetzt die Quellsprache in eine Zielsprache. Dabei wird diese Übersetzung oft stufenweise durchgeführt, sodass sie als Komposition mehrerer Übersetzungen aufgefasst werden kann. Für einen Korrektheitsbegriff für Übersetzungen ist es daher wünschenswert, dass er sich über die Komposition von Übersetzungen fortsetzt: Wenn die Übersetzungen $\sigma_1 : D_1 \to D_2$ und $\sigma_2 : D_2 \to D_3$ korrekt sind, so sollte auch die Korrektheit von $\sigma_2 \circ \sigma_1$ folgen.

Der Vergleich der *Ausdruckskraft* zweier Programmkalküle kann durchgeführt werden, indem Übersetzungen zwischen beiden Kalkülen angegeben werden und im Anschluss Eigenschaften der Übersetzungen gezeigt werden.

Die *Implementierung* eines (neuen) Programmkonstrukts in Form einer Programmbibliothek kann als Übersetzung aufgefasst werden: Sei L eine Programmiersprache, die ein neues Programmierkonstrukt K in Form einer Programmbibliothek zur Verfügung stellt. Die dabei auftretende Fragestellung ist, ob die Implementierung korrekt ist, d. h. die Spezifikation des neuen Konstrukts erfüllt wird. Manchmal wird dabei die Sichtweise vertreten, dass die Implementierung selbst die Semantik (und daher auch die Spezifikation) darstellt. Dieser Ansatz hat jedoch Schwächen, da die Korrektheit der Spezifikation in diesem Fall genauso schwer zu verifizieren ist wie die Korrektheit der Implementierung. Ein weitaus stärkerer Ansatz besteht

darin, zunächst die Sprache L formal um das Konstrukt K (Erweiterung der Syntax und der (operationalen) Semantik) zu erweitern. Sei L' diese Erweiterung. Im Anschluss kann die Implementierung von K in L verwendet werden, um die Übersetzung von L' in L zu definieren. Die Spezifikation stellt daher die Sprache L' dar und die Korrektheit kann anhand von Eigenschaften der Übersetzung nachgewiesen werden.

Die *Erweiterung* einer Programmiersprache um neue Programmkonstrukte (die nicht implementiert werden können, da sie Ausdruckskraft hinzufügen) kann als Übersetzung der alten Sprache in die erweiterte Sprache aufgefasst werden. Hier wäre beispielsweise die Erweiterung einer sequentiellen Programmiersprache um Nebenläufigkeit zu nennen. In diesem Kontext steht nicht die Korrektheit im Vordergrund, allerdings kann die Frage gestellt werden, inwiefern sich die Gleichheiten des ursprünglichen Kalküls auf die Erweiterung übertragen lassen.

In (Schmidt-Schauß et al., 2008b) wurden die folgenden Eigenschaften von Übersetzungen herausgearbeitet:

Definition 3.1.2. *Sei $\sigma : D \to D'$ eine Übersetzung zwischen Programmkalkülen $D = (\mathcal{T}, \mathcal{E}, \mathcal{C}, \mathcal{O})$ und $D' = (\mathcal{T}', \mathcal{E}', \mathcal{C}', \mathcal{O}')$ entsprechend Definition 3.1.1.*

- **Adäquatheit:** *σ ist* adäquat *gdw. für alle $e_1, e_2 \in \mathcal{E}$ gilt: $\sigma(e_1) \leq_{D'} \sigma(e_2) \implies e_1 \leq_D e_2$.*
- **Volle Abstraktheit:** *σ ist genau dann* voll abstrakt, *wenn für alle $e_1, e_2 \in \mathcal{E}$ gilt: $\sigma(e_1) \leq_{D'} \sigma(e_2) \iff e_1 \leq_D e_2$.*
- **Kompositionalität:** *σ ist genau dann* kompositional, *wenn $\sigma(C[e]) = \sigma(C)\sigma(e)$ für alle Kontexte $C \in \mathcal{C}$ und Ausdrücke $e \in \mathcal{E}$ gilt.*
- **Kompositionalität bezüglich Konvergenz:** *σ ist genau dann* kompositional bezüglich Konvergenz, *wenn $\sigma(C[e])\Downarrow'_i \iff \sigma(C)\sigma(e)\Downarrow'_i$ für alle Kontexte $C \in \mathcal{C}$, Ausdrücke $e \in \mathcal{E}$ und Beobachtungsprädikate $\Downarrow_i \in \mathcal{O}$ gilt.*
- **Konvergenzäquivalenz:** *σ ist genau dann* konvergenzäquivalent, *wenn für alle $e \in \mathcal{E}$ und $\Downarrow_i \in \mathcal{O}$ gilt: $e\Downarrow_i \iff \sigma(e)\Downarrow'_i$.*
- **Beobachtungskorrektheit:** *σ ist genau dann* beobachtungskorrekt, *wenn sie kompositional bezüglich Konvergenz und konvergenzäquivalent ist, d. h. für alle $C \in \mathcal{C}, e \in \mathcal{E}, \Downarrow_i \in \mathcal{O}$ stets gilt: $C[e]\Downarrow_i \iff \sigma(C)[\sigma(e)]\Downarrow_i$.*

Die wichtigste Eigenschaft zur Korrektheit einer Übersetzung ist die Beobachtungskorrektheit, denn sie sichert zum einen zu, dass die Konvergenz der Ausdrücke unverändert bleibt (Konvergenzäquivalenz), d. h. erfolgreich reduzierende Programme verbleiben erfolgreich reduzierende Programme, aber auch divergierende Programme verbleiben divergierend, was insbesondere deswegen wichtig ist, da i. A. Fehler mit Divergenz identifiziert werden. Zum anderen sichert die Kompositionalität bzgl. Konvergenz zu, dass die Tests zur kontextuellen Gleichheit (das Prüfen der Konvergenz in Kontexten) übertragbar sind.

In (Schmidt-Schauß et al., 2008b) wurde zudem der Satz bewiesen:

Satz 3.1.3. *Jede beobachtungskorrekte Übersetzung ist auch adäquat.*

Die Adäquatheit der Übersetzung sichert dabei zu, dass die Übersetzung alle bestehenden Ungleichheiten erhält – ungleiche Ausdrücke dürfen nach der Übersetzung nicht als gleich identifiziert werden. Bezogen auf die Äquivalenzklassen bezüglich der kontextuellen Gleichheit bedeutet dies, dass die Klassen zwar durch die Übersetzung zerfallen dürfen, jedoch nicht vereint werden dürfen. Volle Abstraktheit der Übersetzung geht darüber hinaus und fordert, dass die Äquivalenzklassen genau erhalten bleiben. Volle Abstraktheit ist zwar eine wünschenswerte Eigenschaft, sie ist im Allgemeinen jedoch schwer zu erreichen.

Betrachten wir die Anwendungsbereiche für Übersetzungen erneut. Beim Kompilieren ist Adäquatheit ausreichend (neben der Beobachtungskorrektheit), da durchgeführte korrekte Programmtransformationen nach der Übersetzung auch korrekt aus Sicht der Quellsprache sind. Betrachtet man die Übersetzung einer Quellsprache direkt in die Interpretersprache (als Zielsprache des Compilierens) und der Interpreter nimmt keinerlei Optimierung vor, sondern wird nur als Evaluator verwendet, so reicht im Grunde der Nachweis der Konvergenzäquivalenz der Übersetzung aus. Die stufenweise Übersetzung ist auch unproblematisch, denn sämtliche Eigenschaften aus Definition 3.1.2 setzen sich auf der Komposition von Übersetzungen fort:

Satz 3.1.4 ((Schmidt-Schauß et al., 2008b))**.** *Seien D_1, D_2, D_3 Programmkalküle und $\sigma_1 : D_1 \to D_2$ und $\sigma_2 : D_2 \to D_3$ Übersetzungen entsprechend Definition 3.1.1. Für jede der Eigenschaften P von Übersetzungen aus Definition 3.1.2 gilt: Wenn σ_1 und σ_2 beide die Eigenschaft P erfüllen, dann besitzt auch $\sigma_2 \circ \sigma_1$ die Eigenschaft P.*

Vergleicht man Programmkalküle hinsichtlich ihrer Ausdruckskraft, so sollte man diese als ausdrucksgleich bezeichnen, wenn es voll abstrakte Übersetzungen zwischen beiden Sprachen gibt. Semantische Identität der Sprachen kann geschlossen werden, wenn ein *Isomorphismus* bezüglich der Äquivalenzklassen zwischen den Sprachen existiert, also eine bijektive, voll abstrakte Übersetzung existiert.

Die Implementierung von Sprachkonstrukten sollte als korrekt bezeichnet werden, wenn die durch die Implementierung definierte Übersetzung beobachtungskorrekt und daher auch adäquat ist. Da auch die Einbettung als Identitätsfunktion existiert, kann manchmal der folgende in (Schmidt-Schauß et al., 2008b) bewiesene (in (Schmidt-Schauß et al., 2008a) verbesserte) Satz verwendet werden, um sogar volle Abstraktheit der Übersetzung zu folgern:

Satz 3.1.5. *Wenn D' den Kalkül D* erweitert, *d. h. es gibt eine Einbettung $\iota : D \to D'$, die D-Ausdrücke auf D'-Ausdrücke, D-Typen auf D'-Typen, D-Kontexte auf D'-Kontexte abbildet (wobei sämtliche Abbildungen typgerecht sind); und ι ist kompositional, konvergenzäquivalent, injektiv auf Typen, Kontexten, Ausdrücken, dann ist eine beobachtungskorrekte Übersetzung $\sigma : D' \to D$ voll abstrakt, wenn $\sigma \circ \iota$ die Identität auf D-Ausdrücken, D-Kontexten, D-Typen ist und σ injektiv auf den Typen ist. Ebenso ist ι in diesem Fall voll abstrakt.*

Für Erweiterungen einer Programmiersprache um neue (ausdrucksstärkere) Programmierkonstrukte ist es nicht sinnvoll einen Korrektheitsbegriff im Allgemeinen zu definieren, da

neue Konstrukte die Semantik verändern können – eben zu einer neuen Sprache führen können. Sei D der Ursprungskalkül und D' der erweiterte Kalkül. Eine wünschenswerte Eigenschaft ist, dass gleiche Ausdrücke von D in D' gleich verbleiben, da in diesem Fall Gleichheitsnachweise für D-Ausdrücke weiterhin für D' verwendet werden können. Man spricht in diesem Fall von einer *konservativen Erweiterung*, die man formal definieren kann als: Die natürliche Einbettung $\iota : D \to D'$ (welche im Grunde die Identitätsfunktion ist), ist konservativ, wenn für alle D-Ausdrücke e_1, e_2 gilt: $e_1 \sim_D e_2 \implies \iota(e_1) \sim_{D'} \iota(e_2)$. Oft kann man aus der Konservativität der natürlichen Einbettung mit einfachen Mitteln auch folgern, dass ι voll abstrakt ist.

3.2 Anwendungsbeispiele

3.2.1 Die Paarkodierung von Church

Als kleines illustratives Anwendungsbeispiel wurde in (Schmidt-Schauß et al., 2008b) ein call-by-value Lambda-Kalkül mit Paaren, Fixpunktoperator und dämonischer Auswahl in den entsprechenden Kalkül ohne Paare mithilfe der sogenannten Church-Kodierung für Paare (Church, 1941) übersetzt. Paare sind durch den Paar-Konstruktor (v_1, v_2) und die Selektoren **fst** und **snd** im Kalkül mit Paaren repräsentiert, wobei für Paarkomponenten zunächst nur Werte erlaubt wurden. Die Church-Kodierung ϕ ist die Identität auf allen Konstrukten (homomorph über der Termstruktur), außer für die Fälle

$$\begin{array}{lcl} \phi(v_1, v_2) & := & \lambda x.(x\ \phi(v_1)\ \phi(v_2)) \\ \phi(\mathbf{fst}) & := & \lambda p.(p\ (\lambda x, y.x)) \\ \phi(\mathbf{snd}) & := & \lambda p.(p\ (\lambda x, y.y)) \end{array}$$

Die Ergebnisse dazu lassen sich wie folgt zusammenfassen: Die Übersetzung ϕ ist nicht-korrekt, wenn Quell- und Zielkalkül beide ungetypt sind, da in diesem Fall die Konvergenzäquivalenz verletzt wird. Ist der Quellkalkül (einfach-) getypt, und der Zielkalkül ungetypt, so ist die Übersetzung ϕ hingegen korrekt, da für diesen Fall die Beobachtungskorrektheit gezeigt werden konnte. Wenn beide Kalküle einfach- (oder auch Hindley-Milner-) getypt sind, so ist Übersetzung ϕ nicht wohl-definiert, da in diesem Fall getypte Ausdrücke in nicht-typisierbare Ausdrücke übersetzt würden.

Des Weiteren wurden Varianten des Paarkalküls betrachtet: Zum einen ein Kalkül, der beliebige Ausdrücke als Paarkomponenten erlaubt. Hierbei wurde gezeigt, dass die Übersetzung in den Kalkül mit auf Werte beschränkten Paaren voll abstrakt ist. Zum anderen wurde der Paarkalkül mit allgemeinen Paaren mit einer flexibleren Reduktionsstrategie betrachtet, in der beliebige Reihenfolgen in der Auswertung der Paarkomponenten und der Argumente des choice-Operators erlaubt sind. Die Übersetzung in den vorangegangenen Kalkül mit Paaren und deterministischer Auswertung wurde ebenfalls als voll abstrakt nachgewiesen. Mithilfe von Satz 3.1.4 folgt daraus auch, dass die Übersetzung des allgemeinsten Kalküls in den

eingeschränkten Kalkül ohne Paare als Komposition der definierten Übersetzungen ebenfalls adäquat ist.

3.2.2 Korrektheit der Implementierung von Nebenläufigkeitsprimitiven

Im Rahmen des *call-by-value Lambda-Kalküls mit Futures* (Niehren et al., 2006, 2007) wurde in (Schwinghammer et al., 2009b) die Korrektheit der Implementierung verschiedener Primitiven der nebenläufigen Programmierung untersucht. Dies ist ein sehr realistisches Beispiel, da in Programmiersprachen häufig ein Basiskonstrukt zur Synchronisation eingebaut wird (beispielsweise Locks), und andere Primitive (beispielsweise Semaphoren) als Programmbibliothek basierend auf dem Basiskonstrukt zur Verfügung gestellt werden.

Der call-by-value Lambda-Kalkül mit Futures modelliert die Kernsprache von Alice ML[1] (Rossberg et al., 2006). Er ist ein nebenläufiger Kalkül, der Prozesse auswertet. Wir beschreiben kurz die zweistufige Syntax des Kalküls in der Variante namens $\lambda(\mathsf{fh})$. Der Kalkül ist ungetypt. Auf oberster Ebene besitzt die Syntax Prozesse, die wiederum als Unterterme funktionale Ausdrücke enthalten. Die Syntax für Prozesse ist durch die Grammatik

$$P, P_i \in \mathit{Proc} ::= P_1 \mid P_2 \mid \nu x.P \mid x \,\mathsf{c}\, v \mid x \Leftarrow e \mid x \overset{\mathit{susp}}{\Longleftarrow} e \mid x \,\mathsf{h}\, y \mid x \,\mathsf{h}\, \bullet$$

festgelegt, wobei x, y Namen (aus einer unendlichen Menge von Namen) sind, e sind funktionale Ausdrücke und v sind Werte. Wir erläutern die einzelnen Prozesskomponenten:

- $P_1 \mid P_2$ ist die *parallele Komposition* der Prozesse P_1 und P_2.
- Die *Restriktion* $\nu x.P$ schränkt den Geltungsbereich des Namens x auf den Prozess P ein.
- Das Konstrukt $x \Leftarrow e$ ist eine *nebenläufige Future*, die nebenläufige Threads darstellen: Der Thread wertet dabei den Ausdruck e aus und erlaubt den Zugriff auf das Ergebnis über den Namen x. Global wirken die Futures zudem wie rekursive Bindungen, d. h. in $x_1 \Leftarrow e_1 \mid x_2 \Leftarrow e_2 \mid \ldots \mid x_n \Leftarrow e_n$ darf jedes x_i in allen e_j verwendet werden.
- *Lazy Futures* $x \overset{\mathit{susp}}{\Longleftarrow} e$ besitzen die gleichen Eigenschaften wie Futures mit dem Unterschied, dass die Auswertung des Threads zunächst nicht begonnen wird, sondern erst dann, wenn ein anderer Thread den Wert von x anfordert, und damit die Auswertung des Threads anstößt.
- Das Konstrukt $x \,\mathsf{c}\, v$ repräsentiert einen *Speicherplatz*. Dabei ist x der Name des Speicherplatzes und v (ein Wert) ist der Inhalt des Speichers.
- $x \,\mathsf{h}\, y$ ist ein *Handle*, wobei x der Name des Handles ist und y ist der Name einer Future, die allerdings noch nicht existiert. Durch das sogenannte *Binden* eines Handles, wird die Future $y \Leftarrow e$ erzeugt, und der Handle wird zum *verbrauchten Handle* $x \,\mathsf{h}\, \bullet$. Handles dienen im Wesentlichen als Synchronisationskonstrukt: Futures warten auf den Wert

[1] *http://www.ps.uni-saarland.de/alice/*

Kalkül	Typisierung	Datenkonstruktoren	Handles	Puffer
$\lambda(\mathsf{fh})$	×	×	✓	×
$\lambda^\tau(\mathsf{fc})$	✓	✓	×	×
$\lambda^\tau(\mathsf{fch})$	✓	✓	✓	×
$\lambda^\tau(\mathsf{fcb})$	✓	✓	×	✓
$\lambda^\tau(\mathsf{fchb})$	✓	✓	✓	✓

Tabelle 3.1: Übersicht über die Varianten des Lambda-Kalküls mit Futures

der Future y und blockieren solange, bis ein anderer Thread den Handle bindet und dadurch die Future zur Verfügung stellt. Da ein Handle nur einmalig gebunden werden kann, verhalten sich Handles ähnlich zu single-assignment Variablen in imperativen Programmiersprachen, also Speicherplätzen denen einmalig ein Wert zu gewiesen werden kann.

Ausdrücke (und die Teilmenge der Werte) kommen in Futures, lazy Futures, und Speicherplätzen vor. Diese umfassen neben den üblichen Konstrukten des Lambda-Kalküls, Operatoren zur Erzeugung von Futures, lazy Futures, Speicherplätzen und Handles, sowie den Operator $\mathbf{exch}(z, v)$, der den Inhalt der Speicherzelle namens z atomar durch den Wert v ersetzt und den ursprünglichen Inhalt der Zelle als Ergebnis liefert.

Die operationale Semantik ist durch eine small-step-Reduktion definiert (siehe (Niehren et al., 2006, 2007)), die kleinschrittige Ersetzungen in call-by-value Reihenfolge vornimmt. Erfolgreich ist ein Prozess erst dann, wenn sämtliche Futures (mit der Ausnahme der lazy Futures) an einen Wert (also eine Abstraktion oder eine Variable) gebunden sind, und es keinen Zyklus von Namen der Form $x_1 \Leftarrow x_2 \mid \ldots x_{n-1} \Leftarrow x_n \mid x_n \Leftarrow x_1$ zwischen diesen Futures gibt. Dies beschreibt die Menge der Antworten des Kalküls.

In (Schwinghammer et al., 2009b) wurden weitere Varianten betrachtet, deren Unterschiede in Tabelle 3.1 zusammengefasst dargestellt sind. Während $\lambda(\mathsf{fh})$ ungetypt ist, sind alle anderen Varianten mit einem monomorphen Typsystem ausgestattet sind, so dass keine Typfehler zur Laufzeit entstehen können. Der Kalkül $\lambda^\tau(\mathsf{fch})$ ist monomorph getypt und erweitert die Syntax der Ausdrücke gegenüber $\lambda(\mathsf{fh})$ um Datenkonstruktoren und case-Ausdrücke. Der Kalkül $\lambda^\tau(\mathsf{fc})$ gleicht dem Kalkül $\lambda^\tau(\mathsf{fch})$ jedoch wurden Handles entfernt, d. h. die syntaktischen Konstrukte $x\ \mathsf{h}\ \bullet$ und $x\ \mathsf{h}\ y$ sind nicht vorhanden.

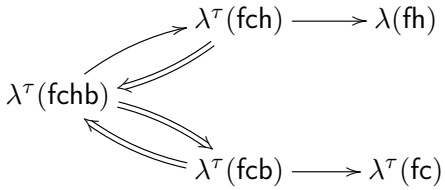
Der Kalkül $\lambda^\tau(\mathsf{fcb})$ ist die Erweiterung des Kalküls $\lambda^\tau(\mathsf{fc})$ um synchronisierende Pufferspeicher $x\ \mathsf{b}\ v$ und $x\ \mathsf{b}\ -$. Das Konstrukt $x\ \mathsf{b}\ v$ ist dabei ein *gefüllter Puffer* namens x mit Inhalt v, $x\ \mathsf{b}\ -$ stellt einen *leeren Puffer* namens x dar. Innerhalb der Ausdrücke gibt es neben Operatoren zum Erzeugen der Puffer die Primitiven $\mathbf{get}(x)$, um den Inhalt aus dem Puffer zu lesen und den Puffer zu leeren, und $\mathbf{put}(x, v)$, um den Puffer namens x mit dem Inhalt v zu füllen. Sowohl **get** als auch **put** führen zum Blockieren des Threads, wenn der Puffer leer bzw. bereits gefüllt ist. Daher eignen sich diese Puffer zur Synchronisation von nebenläufigen Prozessen. Der Kalkül $\lambda^\tau(\mathsf{fchb})$ erweitert den Kalkül $\lambda^\tau(\mathsf{fcb})$ um Handles.

Als Gleichheitsbegriff wurde in allen betrachten Kalkülen die kontextuelle Gleichheit basierend auf May- und Should-Konvergenz verwendet. In (Niehren et al., 2007) wurde die Korrektheit bzgl. dieser Gleichheit für einige Programmtransformationen im Kalkül $\lambda(\mathsf{fh})$ gezeigt.

Die Fragestellung und Motivation der Untersuchung in (Schwinghammer et al., 2009b) ist, ob synchronisierende Puffer mittels Handles (und primitiven Speicherzellen) kodierbar sind, und ebenso die Gegenfrage, ob Handles mittels Pufferspeichern kodierbar sind. Dabei steht im Vordergrund, diese Kodierbarkeiten formal auf Grundlage der Semantik nachzuweisen.

Hierfür wurden im Kalkül $\lambda^{\tau}(\mathsf{fch})$ eine Implementierung für Pufferspeicher und im Kalkül $\lambda^{\tau}(\mathsf{fcb})$ eine Implementierung für Handles entworfen. Um die Korrektheitsbegriffe für Übersetzungen zu verwenden, wurden die Implementierungen formal als Übersetzungen vom erweiterten Kalkül $\lambda^{\tau}(\mathsf{fchb})$ in den entsprechenden Unterkalkül definiert. In (Schwinghammer et al., 2009b) konnte gezeigt werden, dass beide Übersetzungen beobachtungskorrekt und daher auch adäquat sind, die Implementierungen sind daher korrekt.

U. a. um bekannte Programmgleichheiten aus $\lambda(\mathsf{fh})$ in die mächtigeren Kalküle zu importieren, wurde eine Übersetzung zum Wegkodieren von case und Datenkonstruktoren (von $\lambda^{\tau}(\mathsf{fch})$ in $\lambda(\mathsf{fh})$) als beobachtungskorrekt gezeigt. Des Weiteren wurde gezeigt, dass im Kalkül $\lambda^{\tau}(\mathsf{fc})$ ohne Puffer und Handles eine korrekte Implementierung von Puffern existiert, indem die Beobachtungskorrektheit einer Übersetzung $\lambda^{\tau}(\mathsf{fcb}) \to \lambda^{\tau}(\mathsf{fc})$ gezeigt wurde. Durch weitere Einbettungen konnte das gleiche Resultat für $\lambda^{\tau}(\mathsf{fch})$ abgeleitet werden: Auch Handles können durch adäquate Übersetzungen in $\lambda^{\tau}(\mathsf{fc})$ kodiert werden. Insgesamt ergab sich als Resultat das folgende Bild:



Dabei stellen doppelte Pfeile eine voll abstrakte Übersetzung und einfache Pfeile eine adäquate Übersetzung dar. Alle Übersetzungen wurden in (Schwinghammer et al., 2009b) als beobachtungskorrekt gezeigt. Durch Komposition der Übersetzungen können weitere Übersetzungen konstruiert und deren Eigenschaften gefolgert werden.

Als Fazit lässt sich zusammenfassen, dass Pufferspeicher und Handles austauschbare Primitive zur nebenläufigen Programmierung sind, und dies auch formal nachgewiesen werden kann. Die Adäquatheit der Übersetzung von $\lambda^{\tau}(\mathsf{fcb})$ in $\lambda^{\tau}(\mathsf{fc})$ zeigt sogar, dass Zellen mit atomarer **exch**-Operation ausreichend sind. Allerdings ist hierbei anzumerken, dass die Kodierung von Puffern in $\lambda^{\tau}(\mathsf{fc})$ eine busy-wait Implementierung ist, d. h. blockierte Threads warten, indem sie dauernd prüfen, ob sie weiterhin blockiert sind. Als weiteres Resultat ist zu bemerken, dass sich durch die Übersetzungen von $\lambda^{\tau}(\mathsf{fchb})$ hin zur einfacheren Sprache $\lambda(\mathsf{fh})$ viele in (Niehren et al., 2007) bewiesene Gleichheiten auf einfache Weise für $\lambda^{\tau}(\mathsf{fchb})$ wiederverwenden ließen *ohne* erneut aufwändige Korrektheitsbeweise für Programmtransformationen führen zu müssen.

4

Hilfsmittel zum Nachweis kontextueller Äquivalenz

In diesem Kapitel diskutieren wir wesentliche Aspekte zum syntaktischen Nachweis der kontextuellen Äquivalenz. Zum einen fassen wir die Ergebnisse aus (Schmidt-Schauß & Sabel, 2010b) zusammen, welches einen allgemeinen Rahmen zum Nachweis der Gültigkeit eines Kontextlemmas liefert. Ein solches Kontextlemma erleichtert Beweise kontextueller Gleichheiten, da nur eine eingeschränkte Menge von Kontexten (gegenüber allen Kontexten) in Betracht gezogen werden muss. In einem zweiten Teil des Kapitels erörtern wir die sogenannte Diagrammmethode zum Korrektheitsnachweis von Programmtransformationen und präsentieren die Ergebnisse aus (Rau et al., 2012), die einen großen Schritt hin zu Automatisierung solcher Nachweise darstellen.

4.1 Kontextlemmata für Programmkalküle höherer Ordnung

Sei $\leq_P$ die kontextuelle Präordnung eines Programmkalküls bezüglich eines Beobachtungsprädikates P, die definiert sei als $e_1 \leq_P e_2$ genau dann, wenn gilt: $\forall C \in \mathcal{C} : P(C[e_1]) \implies P(C[e_2])$ (wir betrachten hier den ungetypten Fall, der getypte Fall ist analog). Ein *Kontextlemma* sagt aus, dass es ausreicht eine kleinere Menge von Kontexten zu betrachten, um kontextuelle Präordnung zu schließen. Sei $\mathcal{R}$ eine Menge von Kontexten mit $\mathcal{R} \subset \mathcal{C}$, dann ist ein Kontextlemma die Aussage:

$$\forall R \in \mathcal{R} : P(R[e_1]) \implies P(R[e_2]) \text{ ist äquivalent zu } e_1 \leq_P e_2.$$

In vielen Fällen ist $\mathcal{R}$ gerade die Menge der Reduktionskontexte, wie wir sie in den Beispielen in Abschnitt 2.1.1 definiert haben.

Die Existenz eines Kontextlemmas erleichtert die Beweise von kontextuellen Gleichheiten und die Nachweise der Korrektheit von Programmtransformationen erheblich, da wesentlich weniger Kontexte und damit Fälle betrachtet werden müssen.

Für viele Kalküle wurde ein solches Kontextlemma nachgewiesen (z. B. (Milner, 1977; Gordon, 1999; Niehren et al., 2007; Schmidt-Schauß et al., 2008c; Sabel & Schmidt-Schauß, 2008, 2011b)). Ähnliche Lemmas sind sogenannte CIU-Lemmas[1], die z. B. in (Mason & Talcott, 1991; Felleisen & Hieb, 1992; Mason et al., 1996; Pitts & Stark, 1998; Lassen, 1998) für verschiedene call-by-value Lambda-Kalküle nachgewiesen wurden. Für eine Klasse von solchen Kalkülen wurde in (Ford & Mason, 2001, 2003) das CIU-Lemma formal mithilfe eines automatischen Theorembeweisers als korrekt gezeigt. Gegenüber dem oben umrissenen Kontextlemma, verwenden CIU-Lemmas zusätzlich zu Reduktionskontexten schließende Substitutionen, um ausschließlich die Konvergenz geschlossener Ausdrücke zu vergleichen.

Das wesentliche Ziel der Untersuchung in (Schmidt-Schauß & Sabel, 2010b) ist der Nachweis eines generellen Kontextlemmas für call-by-need Lambda-Kalküle mit rekursiven Let-Ausdrücken, wobei auch nichtdeterministische Kalküle mit eingeschlossen werden und daher alle drei Beobachtungsprädikate May-, Should-, und Must-Konvergenz betrachtet werden. Tatsächlich gehen die Resultate in (Schmidt-Schauß & Sabel, 2010b) darüber hinaus, denn der letztlich verwendete Rahmen umfasst auch Kalküle mit call-by-value und call-by-name Reduktion und nicht ausschließlich Lambda-Kalküle, auch für Prozesskalküle wie den π-Kalkül (Milner, 1999; Sangiorgi & Walker, 2001) oder den Join-Kalkül (Fournet & Gonthier, 1996; Laneve, 1996) kann der Rahmen ein Kontextlemma liefern.

Wir erläutern zunächst die Resultate aus (Schmidt-Schauß & Sabel, 2010b) für sogenannte *sharende Programmkalküle*. Diese Kalküle führen keine oder nur wenige überflüssige Ersetzungen während der Reduktion durch. Die in Abschnitt 2.1.1 vorgestellten Lambda-Kalküle erfüllen diese Bedingung nicht (sie sind daher keine sharenden Programmkalküle), da sie die β-Reduktion $(\lambda x.e_1)\ e_2 \to e_1[e_2/x]$ verwenden, welche den Ausdruck e_2 an beliebige Positionen innerhalb e_1 kopiert und dadurch vermehrfacht. Ein Kalkül mit Sharing könnte stattdessen ein `let`-Konstrukt verwenden und β-Reduktion durch eine sharende Variante der β-Reduktion und eine Kopierregel ersetzen: Die Anwendung einer Abstraktion auf ein Argument wird dabei in einen `let`-Ausdruck überführt, d. h. $(\lambda x.e_1)\ e_2 \to \texttt{let}\ x = e_2\ \texttt{in}\ e_1$, und die Kopierregel $\texttt{let}\ x = v\ \texttt{in}\ R[x] \to \texttt{let}\ x = v\ \texttt{in}\ R[v]$ kopiert Werte in eine benötigte Reduktionsposition[2]. Ein solcher `let`-Kalkül ist ein sharender Programmkalkül.

In (Schmidt-Schauß & Sabel, 2010b) werden zwei Arten von sharenden Programmkalkülen unterschieden, für die als Resultat verschiedene Kontextlemmata nachgewiesen werden konnten: Sogenannte *stark-sharende Kalküle* und sogenannte *schwach-sharende Kalküle*. Der Unterschied zwischen beiden Kalkülen liegt darin, dass schwach-sharende Kalküle Variablenersetzungen überall durchführen dürfen, während stark-sharende Kalkül dies ausschließlich an sogenannten Reduktionspositionen durchführen dürfen. Z. B. ist ein Kalkül mit `let` und entsprechender sharender β-Reduktion und Kopierregel stark-sharend, da er keine Variablenersetzungen durchführt. Im Vergleich dazu ist ein Kalkül mit auf Variablen eingeschränkte Argumentposition der Anwendung (d. h. nur Anwendungen der Form $(e\ x)$ sind erlaubt, wobei x

[1] CIU ist eine Abkürzung für „closed instances of use“

[2] Die Definition der Reduktionskontexte muss dementsprechend angepasst werden.

eine Variable ist) und der entsprechend eingeschränkten Beta-Reduktion $((\lambda x.e_1)\ y) \to e_1[y/x]$ ein schwach sharender Kalkül, da Variablen durch Variablen substituiert werden.

Wir beschreiben die Anforderungen an stark- und schwach-sharende Programmkalküle an dieser Stelle informell, die genauen Bedingungen sind (Schmidt-Schauß & Sabel, 2010b) zu entnehmen. Die einfache Beschreibung eines Programmkalküls wie in Definition 2.1.1 reicht nicht aus, um den Beweis eines Kontextlemmas zu führen. Vielmehr wird eine genauere Beschreibung des syntaktischen Aufbaus der Ausdrücke und der Eigenschaften der Standardreduktion benötigt. Daher wird in (Schmidt-Schauß & Sabel, 2010b) die Syntax von Ausdrücken und Antworten eines Programmkalküls mithilfe von sogenannter abstrakter Syntax höherer Ordnung (wie sie z. B. auch in (Howe, 1989, 1996) verwendet wird) definiert. Diese legt genau die Operatoren der Sprache und die bindenden Konstrukte (wie beispielsweise Lambda-Abstraktionen) fest. Auch einfache Typsysteme sind in dieser Syntax erlaubt. Ein erlaubter Spezialoperator (der nicht in dieses Schema passt) ist das `letrec`-Konstrukt, welches auch erlaubt ist. `letrec`-Ausdrücke haben die Form `letrec` $x_1 = e_1 \ \ldots \ x_n = e_n$ `in` e und erlauben dabei, dass die Variablen $x_1, \ldots, x_n$ in allen Ausdrücken e_i und e vorkommen, d. h. die Bindungen $x_1 = e_1, \ldots, x_n = e_n$ dürfen verschränkt rekursiv sein.

Neben dieser genauen Beschreibung der Syntax (für Ausdrücke und Antworten) verlangt der Rahmen in (Schmidt-Schauß & Sabel, 2010b) einen sogenannten Unwind-Algorithmus, der die Reduktionspositionen eines Ausdrucks (oder auch eines Kontextes) in top-down Weise berechnet. D. h. für einen Ausdruck e liefert der Unwind-Algorithmus eine Sequenz von Reduktionspositionen $p_1, \ldots, p_n$ in e. Die *Reduktionskontexte* $\mathcal{R}$ des Programmkalküls sind definiert durch den Unwind-Algorithmus: Jeder Kontext C, dessen Lochposition in der Ausgabe des Unwind-Algorithmus bei Eingabe C erscheint, ist ein Reduktionskontext.

Für Antworten des Programmkalküls muss gelten, dass die Antworteigenschaft nur von Reduktionspositionen abhängt, d. h. wenn ein Unterausdruck einer Antwort an einer Nichtreduktionsposition durch einen beliebigen anderen Ausdruck ausgetauscht wird, bleibt die Antworteigenschaft erhalten.

Die Standardreduktion darf im Wesentlichen nur Reduktionspositionen verändern, für Unterausdrücke an Nichtreduktionspositionen darf die Standardreduktion diese ausschließlich entfernen, vervielfachen, verschieben und umbenennen. Für schwach-sharende Kalküle ist zusätzlich erlaubt, dass Variablen durch Variablen an allen Positionen ersetzt werden dürfen. All diese Eigenschaften müssen stabil bezüglich der Umbenennung gebundener Variablen und Permutationen auf freien Variablen sein.

In die Klasse der stark-sharenden Kalküle fallen beispielsweise die call-by-need Lambda-Kalküle mit `let` aus (Ariola & Felleisen, 1997; Maraist et al., 1998), deterministische, erweiterte `letrec`-Kalkül aus (Schmidt-Schauß et al., 2008c), der nichtdeterministische `let`-Kalkül mit erratischer Auswahl aus (Mann, 2005b), der nichtdeterministische `letrec`-Kalkül mit dem amb-Operator aus (Sabel & Schmidt-Schauß, 2008) und der call-by-value Prozesskalkül mit Futures aus (Niehren et al., 2007). Weakly-sharende Kalküle mit `let` bzw. `letrec` sind beispielsweise in (Moran, 1998; Moran et al., 2003) zu finden. Die wesentliche Reduktion des π-Kalküls (Milner, 1999; Sangiorgi & Walker, 2001) ist die Kommunikationsregel zwischen Prozessen. Sie hat

die Form $x(y).P \mid \overline{x}z.Q \to P[z/y] \mid Q$. Da eine volle Substitution der Variablen x durch z in P stattfindet, handelt es sich um einen schwach-sharenden Kalkül.

Für $P \in \{\downarrow, \Downarrow, \Downarrow\!\!\Downarrow\}$ sei $\leq_P$ die kontextuelle Präordnung in einem solchen Kalkül bezüglich May-, Should- und Must-Konvergenz. Sei $\leq_{P,\nu}$ für $P \in \{\downarrow, \Downarrow, \Downarrow\!\!\Downarrow\}$ die kontextuelle Präordnung mit Variablenersetzungen definiert durch

$$e_1 \leq_{P,\nu} e_2 \text{ gdw. } \forall C \in \mathcal{C}, \forall \text{Variablenersetzungen } \nu : P(C[\nu(e_1)]) \implies P(C[\nu(e_2)]).$$

Die kontextuelle Präordnung mit Variablenersetzungen ist im Allgemeinen eine etwas feinere Gleichheit als die kontextuelle Präordnung ohne diese Ersetzungen. In schwach-sharenden Kalkülen treten beide Varianten in Untersuchungen zu solchen Programmkalkülen auf, und daher wurden in (Schmidt-Schauß & Sabel, 2010b) beide Varianten betrachtet.

Die Relationen $\leq_{P,\mathcal{R}}$ und $\leq_{P,\mathcal{R}\nu}$ für $P \in \{\downarrow, \Downarrow, \Downarrow\!\!\Downarrow\}$ seien die kontextuellen Präordnungen, wobei die Kontexte in der Definition auf Reduktionskontexte eingeschränkt sind.

In (Schmidt-Schauß & Sabel, 2010b) wurden die folgenden Kontextlemmata bewiesen:

Theorem 4.1.1. *Für stark-sharende Kalküle gilt:*

- $\leq_{\downarrow,\mathcal{R}} = \leq_{\downarrow}$
- $\leq_{\Downarrow\!\!\Downarrow,\mathcal{R}} = \leq_{\Downarrow\!\!\Downarrow}$
- $\leq_{\downarrow,\mathcal{R}} \cap \leq_{\Downarrow,\mathcal{R}} = \leq_{\downarrow} \cap \leq_{\Downarrow}$

Für schwach-sharende Kalküle gilt:

- $\leq_{\downarrow,\mathcal{R}\nu} = \leq_{\downarrow\nu} \subseteq \leq_{\downarrow}$
- $\leq_{\Downarrow\!\!\Downarrow,\mathcal{R}\nu} = \leq_{\Downarrow\!\!\Downarrow,\nu} \subseteq \leq_{\Downarrow\!\!\Downarrow}$
- $\leq_{\downarrow,\mathcal{R}\nu} \cap \leq_{\Downarrow,\mathcal{R}\nu} = \leq_{\downarrow\nu} \cap \leq_{\Downarrow\nu} \subseteq \leq_{\Downarrow}$

Das Theorem zeigt daher, dass in stark-sharenden Kalkülen die Betrachtung der Reduktionskontexte zum Nachweis oder zur Widerlegung der kontextuellen Gleichheit zweier Ausdrücke ausreichend ist. Dies gilt für die kontextuellen Präordnungen bezüglich May-Konvergenz und die Must-Konvergenz, für die Präordnung bezüglich Should-Konvergenz muss stets der Schnitt mit der Präordnung bezüglich May-Konvergenz betrachtet werden. Für schwach-sharende Kalküle zeigt das Theorem die analoge Aussage, wobei stets alle Variablenersetzungen mitbetrachtet müssen. Zudem zeigen die Inklusionen, dass die Einbeziehung der Variablenersetzungen in die Präordnungen, diese verfeinert.

Basierend auf diesen Resultaten konnten in (Schmidt-Schauß & Sabel, 2010b) ebenfalls sogenannte CIU-Lemmas für Kalküle hergeleitet werden, die weder stark- noch schwach-sharend sind. Dabei wurden die Relationen $\leq_{P,\mathcal{R}\sigma}$ betrachtet, die analog zu den Relationen $\leq_{P,\mathcal{R}\nu}$ definiert sind, wobei anstelle der Variablenersetzungen Substitutionen σ treten, die Variablen durch beliebige Ausdrücke (bzw. durch Werte, für call-by-value Kalküle) ersetzen. Die entsprechenden sogenannten *nicht-sharenden Kalküle* müssen in diesem Fall Bedingungen erfüllen, die sich grob dadurch charakterisieren lassen, dass die Standardreduktion einer Variante

der β-Reduktion gleicht, die Variablen durch Ausdrücke (bzw. durch Werte, für call-by-value Kalküle) ersetzt, die Reduktion darf nicht unterhalb von Bindern stattfindens und die Kalküle haben kein `letrec`-Konstrukt. Sämtliche Kalküle aus Abschnitt 2.1.1 fallen in die Klasse der nicht-sharenden Kalküle. Unter Zuhilfenahme der Methoden zur Korrektheit von Übersetzungen (siehe Abschnitt 3.1) wurde in (Schmidt-Schauß & Sabel, 2010b) das folgende Kontextlemma für nicht-sharende Kalküle nachgewiesen:

Theorem 4.1.2. *Für nicht-sharende Kalküle gilt:*

- $\leq_{\downarrow,\mathcal{R}\sigma} \subseteq \leq_{\downarrow}$
- $\leq_{\downarrow,\mathcal{R}\sigma} \cap \leq_{\Downarrow,\mathcal{R}\sigma} \subseteq \leq_{\downarrow} \cap \leq_{\Downarrow}$
- $\leq_{\Downarrow,\mathcal{R}\sigma} \subseteq \leq_{\Downarrow}$

Die wesentliche Idee des Beweises besteht darin, den nicht-sharenden Kalkül um ein Konstrukt zum Sharing (d. h. ein `let`) zu erweitern und die Reduktion in diesem erweiterten Kalkül derart anzupasssen, dass dieser ein sharender Kalkül ist. Schließlich wird eine adäquate Übersetzung (siehe Definition 3.1.2) vom erweiterten Kalkül in den ursprünglichen Kalkül verwendet, um das Kontextlemma für sharende Kalküle auf den nicht-sharenden Kalkül zu übertragen.

4.2 Die Diagrammmethode und Automatisierung mithilfe von Terminierungsbeweisern

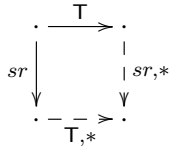
Die Diagrammmethode ist ein syntaktisches Verfahren, um die Korrektheit von Programmtransformationen nachzuweisen. Der Einfachheit halber beschreiben wir die Methode für ungetypte Programmkalküle und beschränken uns auf die May-Konvergenz[3].

Sei $(\mathcal{E}, \mathcal{C}, \xrightarrow{sr}, \mathcal{A})$ ein ungetypter Programmkalkül und $\xrightarrow{\mathsf{T}} \subseteq \mathcal{E} \times \mathcal{E}$ eine Programmtransformation. Für den Korrektheitsnachweis der Programmtransformation $\xrightarrow{\mathsf{T}}$ ist es notwendig zu zeigen, dass $\xrightarrow{\mathsf{T}} \subseteq \sim_{\downarrow}$ gilt. Wir betrachten im Folgenden nur die Teilaussage $\xrightarrow{\mathsf{T}} \subseteq \leq_{\downarrow}$, da die weitere Inklusion $\xrightarrow{\mathsf{T}} \subseteq \geq_{\downarrow}$ analog gezeigt werden kann (durch Zeigen von $\xleftarrow{\mathsf{T}} \subseteq \leq_{\downarrow}$, wobei $\xleftarrow{\mathsf{T}}$ die Inverse der Relation $\xrightarrow{\mathsf{T}}$ ist).

Nach Einsetzen der Definition für die kontextuelle Präordnung reduziert sich der Beweis der Aussage $\xrightarrow{\mathsf{T}} \subseteq \leq_{\downarrow}$ auf: Zeige, dass für alle Kontexte $C \in \mathcal{C}$ und alle Ausdrücke e_1, e_2 mit $e_1 \xrightarrow{\mathsf{T}} e_2$ gilt: $C[e_1]\downarrow \implies C[e_2]\downarrow$. Wenn ein Kontextlemma gilt, kann die Klasse der Kontexte für diesen Beweis entsprechend eingeschränkt werden. Um die Notation zu vereinfachen, nehmen wir an, dass die Transformation $\xrightarrow{\mathsf{T}}$ bereits abgeschlossen bzgl. einer solchen ausreichenden Klasse von Kontexten ist. Daraus ergibt sich, dass $\xrightarrow{\mathsf{T}} \subseteq \leq_{\downarrow}$ gefolgert werden kann, wenn für alle Ausdrücke e_1, e_2 mit $e_1 \xrightarrow{\mathsf{T}} e_2$ die Implikation $e_1\downarrow \implies e_2\downarrow$ gilt. Die Diagrammmethode

[3] Die Methode ist jedoch auch für die Should-Konvergenz anwendbar, siehe (Niehren et al., 2007; Sabel & Schmidt-Schauß, 2008; Sabel, 2008; Sabel & Schmidt-Schauß, 2011b).

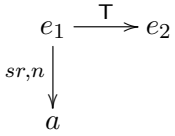
besteht nun aus zwei Schritten: Im ersten Schritt werden alle kritischen Überlappungen zwischen einem Transformationsschritt $e_1 \xrightarrow{\mathsf{T}} e_2$ und einem Standardreduktionsschritt $e_1 \xrightarrow{sr} e_1'$ berechnet und gezeigt, dass das daraus resultierende kritische Paar zusammenführbar ist. Dies ergibt sogenannte Gabeldiagramme der Form



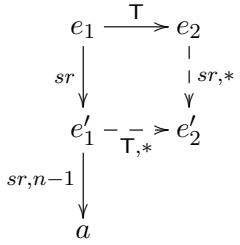
Dabei sind durchgezogene Pfeile die gebenen Reduktionen und Transformationen und gestrichelte Pfeile sind Existenz-quantifiziert – sie bezeugen die Zusammenführbarkeit. Die Diagramme abstrahieren dabei von den konkreten Ausdrücken e_i und stellen lediglich die möglichen Transformations- und Reduktionsschritte abstrakt dar. Dabei dürfen an den Reduktions- und Transformationspfeilen weitere Informationen enthalten sein: Die Standardreduktion ist oft in verschieden benannte Reduktionen aufgeteilt, dieser Name kann im Diagramm vorkommen und dadurch die Fälle einschränken. Auch Variablen für diese Namen, die wie eine Allquantifizierung wirken, sind erlaubt. Ein solches Gabeldiagramm stellt daher eine (u. U. unendliche) Menge von konkreten Überlappungen und Zusammenführbarkeiten dar. In linearer Schreibweise wird obiges Diagramm als $\xleftarrow{sr} \xrightarrow{\mathsf{T}} \rightsquigarrow \xrightarrow{\mathsf{T},*} \xleftarrow{sr,*}$ dargestellt.

Eine Menge von Gabeldiagrammen ist *vollständig* bezüglich einer Transformation $\xrightarrow{\mathsf{T}}$, wenn jede konkrete Überlappung $e_1' \xleftarrow{sr} e_1 \xrightarrow{\mathsf{T}} e_2$ durch mindestens ein Diagramm erfasst wird (wobei auch die entsprechenden Existenz-quantifizierten Reduktionen, Transformationen und Ausdrücke existieren müssen). Neben den Gabeldiagrammen werden noch sogenannte Antwortdiagramme berechnet, die angeben, wie sich die Transformation bezüglich Antworten verhält. Ein Antwortdiagramm hat dabei die Form $A \xrightarrow{\mathsf{T}} \rightsquigarrow A \xleftarrow{sr,*}$. Hierbei repräsentiert A abstrakt eine Antwort aus $\mathcal{A}$.

Der zweite Schritt der Diagrammmethode besteht darin, die vollständigen Sätze von Antwort- und Gabeldiagrammen für $\xrightarrow{\mathsf{T}}$ zu verwenden, um induktiv die Aussage „aus $e_1 \xrightarrow{\mathsf{T}} e_2$ folgt $e_1{\downarrow} \implies e_2{\downarrow}$“ für alle Ausdrücke e_1, e_2 nachzuweisen. Dabei wird zunächst die Existenz einer Folge $a \xleftarrow{sr,n} e_1 \xrightarrow{\mathsf{T}} e_2$ angenommen, die bezeugt, dass e_1 in n-Schritten auf eine Antwort reduziert. Im folgenden Bild ist diese Situation als Gabel dargestellt:



Anschließend wird ein Induktionsbeweis über n (oder auch kompliziertere Induktionsmaße, je nach Art der Diagramme) geführt: Für den Basisfall ($n = 0$) werden die Antwortdiagramme verwendet, um $e_2{\downarrow}$ zu folgern. Für den allgemeinen Fall ($n > 0$) wird auf die oberste Gabel ein Gabeldiagramm angewendet:



Durch Anwenden der Induktionshypothese wird schließlich gefolgert, dass e'_2 und daher auch e_2 konvergiert.

Je nach Art der Diagramme benötigt diese Induktion jedoch stärkere Induktionsbehauptungen (z. B. „für jede konvergierende Reduktionssequenz der Länge n von e_1 aus, gibt es eine Reduktionssequenz für e_2 der Länge $\leq n$“) oder kompliziertere Induktionssmaße, die beispielsweise Termmaße beinhalten.

Die Diagrammmethode wurde in (Kutzner & Schmidt-Schauß, 1998; Kutzner, 2000) eingeführt und in weiteren Kalkülen erfolgreich angewendet (z. B. (Schmidt-Schauß, 2003; Mann, 2005b,a; Schmidt-Schauß & Sabel, 2007; Niehren et al., 2007; Schmidt-Schauß et al., 2008c; Sabel & Schmidt-Schauß, 2008; Sabel, 2008; Schmidt-Schauß & Machkasova, 2008; Sabel et al., 2009; Sabel & Schmidt-Schauß, 2011b)), um Korrektheitsresulate für Programmtransformationen zu erzielen.

Die Beweise sind im Allgemeinen aufwändig, da das Berechnen der Diagramme die Betrachtung vieler Fälle erforder, und innerhalb der Induktion entsprechende Maße zu finden sind, die zeigen, dass das Anwenden der Diagramme terminiert.

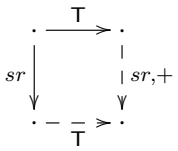
Das Ziel des DFG-Projektes “Automatischer Korrektheitsnachweis von Programmtransformation”[4] ist daher diesen – in den genannten Artikeln per Hand durchgeführten – Korrektheitsnachweis zu automatisieren. Die Berechnung der Diagramme wurde in (Rau & Schmidt-Schauß, 2010, 2011) exemplarisch für zwei Programmkalküle automatisiert, indem ein komplexer Unifikationsalgorithmus entworfen und implementiert wurde.

Das Ziel der Arbeit (Rau et al., 2012) war es, den zweiten Schritt, d. h. die Automatisierung der Induktion, durchzuführen. Hierfür wurden zunächst die Beziehungen zwischen Diagrammen in konkreter (d. h. mit Ausdrücken) und abstrakter (d. h. ohne Ausdrücke, alleinig die Pfeile) Darstellung genau untersucht. Schließlich wurde eine Übersetzung definiert, die Gabel- und Antwortdiagramme als Termersetzungssystem (Baader & Nipkow, 1998) auffasst. Die Objekte dieser Ersetzungssysteme sind Reduktionsfolgen, die aus Transformationsschritten und Standardreduktionen und einem Antwortsymbol bestehen. Die Antwort- und Gabeldiagramme sind Ersetzungsregeln auf diesen Objekten. Es wurde in (Rau et al., 2012) nachgewiesen, dass aus der innermost-Terminierung (Baader & Nipkow, 1998) des Termersetzungssystems folgt, dass ein Induktionsmaß für die Induktion im Diagrammbeweis vorhanden ist, und somit die Korrektheit der Transformation gefolgert werden kann. Durch Verwenden des automatischen Terminierungsbeweisers AProVE[5] (Giesl et al., 2006) kann die Terminierung automa-

[4] *http://www.ki.informatik.uni-frankfurt.de/research/dfg-diagram/en/*
[5] *http://aprove.informatik.rwth-aachen.de/*

tisch bewiesen werden, was exemplarisch für einige Beispiele in (Rau et al., 2012) durchgeführt wurde.

Ein wesentlicher Schritt bei der Übersetzung der Diagramme in abstrakter Form in Termersetzungssysteme war es, die Menge der Regeln *endlich* zu halten. Gabeldiagramme stellen im Allgemeinen eine unendliche Menge von konkreten Überlappungen dar, insbesondere dürfen in den Diagrammen Reduktions- und Transformationspfeile als transitiver Abschluss vorhanden sein (markiert mit $+$). Z. B. ist das folgende Diagramm syntaktisch zulässig:



Durch den transitiven Abschluss auf der rechten Seite, repräsentiert das Diagramm unendlich viele konkrete Diagramme – je nachdem, wie viele konkrete Reduktion für den Pfeil $\xrightarrow{sr,+}$ verwendet werden.

Zum automatisierten Terminierungsnachweis eines Termersetzungssystem ist es erforderlich, dass das entsprechende Termersetzungssystem aus endlich vielen Regeln besteht. Daher mussten solche transitiven Abschlüsse kodiert werden. Hierfür wurden sogenannte Integer-Termersetzungssysteme (Fuhs et al., 2009) verwendet. Diese erlauben es, Zahlen zu raten (auf rechten Seiten von Termersetzungsregeln dürfen freie Variablen vorkommen, die Zahlen repräsentieren). Dementsprechend wurde obiges Diagramm in die Termersetzungsregeln

$$\begin{array}{lcl} T(sr(x)) & \to & w(k,x) \text{ mit } k > 0 \\ w(k,x) & \to & sr(w(k-1,x)) \\ w(1,x) & \to & sr(T(x)) \end{array}$$

kodiert. Die erste Regel ersetzt die Gabel durch ein neues Funktionssymbol mit einer Zahlvariablen k, die Zahl k wird durch die beiden anderen Regeln abgebaut und erzeugt dementsprechend viele $\xrightarrow{sr}$-Reduktionen.

Da AProVE die innermost-Terminierung für Integer-Termersetzungssysteme zeigen kann, konnte diese Kodierung verwendet werden, und damit ein großer Schritt hin zum automatischen Korrektheitsbeweis von Programmtransformationen vollzogen werden.

5

Semantik von Call-by-need Letrec-Kalkülen: Bisimulation, Isomorphie, Typisierung

Erweiterte call-by-need Letrec-Kalküle sind geeignete Modelle für verzögert auswertende funktionalen Programmiersprachen wie beispielweise Haskell (Peyton Jones, 2003). Auch nichtdeterministische Erweiterungen solcher Sprachen sind interessant, da sie z. B. das Ein- und Ausgabeverhalten in solchen Sprachen modellieren können, aber auch als Kernsprache funktional-logischer Sprachen (Antoy & Hanus, 2010) wie z. B. Curry[1] (Hanus et al., 2006) dienen. In diesem Kapitel werden verschiedene semantische Aspekte von solchen Sprachen erörtert. Das Kapitel fasst die Ergebnisse der drei Arbeiten (Schmidt-Schauß et al., 2010b), (Schmidt-Schauß et al., 2011) und (Sabel et al., 2009) zusammen.

In (Schmidt-Schauß et al., 2010b) wird die Korrektheit und Vollständigkeit von applikativer Bisimulation (vgl. Abschnitt 2.2.3) für einen deterministischen Lambda-Kalkül mit rekursiven let-Ausdrücken nachgewiesen. Als weiteres Resultat wird bewiesen, dass dieser Kalkül isomorph bezüglich kontextueller Äquivalenz zum Lazy-Lambda-Kalkül ist, d. h. in diesem Kalkül können die `letrec`-Ausdrücke weg kodiert werden, sie sind im Grunde nur syntaktischer Zucker.

Die Arbeit (Schmidt-Schauß et al., 2011) liefert ein Gegenbeispiel, welches zeigt, dass die applikative Bisimulation in *nichtdeterministischen* letrec-Kalkülen keinen sinnvollen Gleichheitsbegriff liefern kann, da sie weder korrekt noch vollständig bezüglich der kontextuellen Gleichheit in solchen Kalkülen ist. Generell zeigte sich, dass in solchen Kalkülen die sogenannte *Extensionalität* bezüglich der kontextuellen Gleichheit selbst für Abstraktionen falsch ist: Aus der (kontextuellen) Gleichheit von $(\lambda x.e_1)\ r$ und $(\lambda x.e_2)\ r$ für alle Ausdrücke r darf in diesen Kalkülen nicht die Gleichheit von $\lambda x.e_1$ und $\lambda x.e_2$ gefolgert werden. Da die applikative

[1] *http://www.curry-language.org*

Bisimulation im Grunde diese Extensionalitätseigenschaft verwendet, kann sie nicht korrekt sein.

Schließlich werden im letzten Abschnitt die Untersuchungen aus (Sabel et al., 2009) zusammengefasst, die sich mit der Fragestellung beschäftigen, wie kontextuelle Gleichheiten im call-by-need letrec-Kalkül mit seq, case und Datenkonstruktoren unter (prädikativer) *polymorpher Typisierung* gezeigt werden können. Dieser Kalkül modelliert die Kernsprache von Haskell sehr genau.

5.1 Applikative Bisimulation im call-by-need Lambda-Kalkül mit letrec

In (Schmidt-Schauß et al., 2010b) wurde der (ungetypte) call-by-need Kalkül L_{need} mit `letrec` untersucht, der den Lazy-Lambda-Kalkül L_{lazy} (siehe Beispiel 2.1.4) um `letrec`-Ausdrücke erweitert. Die Syntax von Ausdrücken und Antworten ist dementsprechend in Abbildung 5.1 definiert. Kontexte C sind alle Ausdrücke mit Loch $[\cdot]$, die entstehen, wenn ein Unterausdruck eines Ausdrucks durch das Loch ersetzt wird.

Die Standardreduktion verfolgt (anders als im Lazy-Lambda-Kalkül) die call-by-need Strategie und besteht aus den sechs Regeln, die in Abbildung 5.1 gezeigt sind. Die Regeln werden angewendet, nachdem ein Markierungsalgorithmus (der die Suche nach dem nächsten Redex durchführt) den Ausdruck entsprechend mit den Markierungen S, T und V versehen hat. Für einen Ausdruck e startet der Markierungsalgorithmus mit e^T und wendet die Regeln in Abbildung 5.1 solange wie möglich an. Die Regeln der Standardreduktion dürfen dann angewendet werden, wenn die Markierungen passen. Dies definiert die Standardreduktion $\xrightarrow{need}$. Im Gegensatz zur call-by-name Auswertung des Lazy-Lambda-Kalküls werden Doppelauswertungen vermieden, indem anstelle der β-Reduktion die (lbeta)-Reduktion im Zusammenspiel mit den Kopierregeln (cp-in) und (cp-e) verwendet werden, die Werte (Abstraktionen oder Variablen) an benötigte Stellen kopieren. Die Regeln (llet-in), (llet-in) und (lapp) dienen dem Anordnen von verschachtelten `letrec`-Ausdrücken und Applikationen. Da die Standardreduktion eindeutig ist, ist der L_{need}-Kalkül deterministisch. Wir schreiben $\leq_{need}$ für die kontextuelle Präordnung in L_{need} (bzgl. May-Konvergenz) und verwenden $\sim_{need}$ für die kontextuelle Äquivalenz. L_{need} ist ein stark-sharender Kalkül im Sinne von Abschnitt 4.1. Daher gilt das entsprechende Kontextlemma (Theorem 4.1.1) für diesen Kalkül.

Die in (Schmidt-Schauß et al., 2010b) erörterten Fragestellungen sind zum einen, ob in L_{need} applikative Bisimulation mit der kontextuellen Gleichheit zusammenfällt und zum anderen, wie die Beziehung der kontextuellen Gleichheit zur kontextuellen Gleichheit im Lazy-Lambda-Kalkül ist.

Applikative (Bi-)Simulation ist in L_{need} wie folgt definiert:

Definition 5.1.1 (Bisimulation in L_{need})**.** *Sei F_{need} der folgende Operator auf binären Relationen über geschlossenen L_{need}-Ausdrücken. Sei η eine binäre Relation über geschlossenen Ausdrücken, dann gilt $(e_1, e_2) \in F(\eta)$ genau dann, wenn die folgende Bedingung erfüllt ist: Wenn $e_1 \xrightarrow{need,*} e_1'$, wobei e_1' eine L_{need}-WHNF ist, dann gilt $e_2 \xrightarrow{need,*} e_2'$, wobei e_2' eine L_{need}-WHNF ist, und für alle geschlossenen letrec-freien Abstraktionen r und $r := \Omega$ gilt $((e_1'\ r), (e_2'\ r)) \in \eta$*

Ausdrücke: $e, e_i \in \mathcal{E} ::= x \mid \lambda x.e \mid (e_1\ e_2) \mid \texttt{letrec}\ x_1 = e_1, \ldots x_n = e_n\ \texttt{in}\ e$

Antworten: $a \in \mathcal{A} ::= \lambda x.e \mid \texttt{letrec}\ x_1 = e_1, \ldots x_n = e_n\ \texttt{in}\ \lambda x.e$

Standardreduktionen: (v ist eine Abstraktion oder eine Variable)

(lbeta)	$C[((\lambda x.e_1)^S\ e_2)] \to C[(\texttt{letrec}\ x = e_2\ \texttt{in}\ e_1)]$
(cp-in)	$(\texttt{letrec}\ x = v^S, Env\ \ \texttt{in}\ C[x^V]) \to (\texttt{letrec}\ x = v, Env\ \ \texttt{in}\ C[v])$
(cp-e)	$(\texttt{letrec}\ x = v^S, Env, y = C[x^V]\ \texttt{in}\ e) \to (\texttt{letrec}\ x = v, Env, y = C[v]\ \texttt{in}\ e)$
(llet-in)	$(\texttt{letrec}\ Env_1\ \texttt{in}\ (\texttt{letrec}\ Env_2\ \texttt{in}\ e)^S) \to (\texttt{letrec}\ Env_1, Env_2\ \texttt{in}\ e)$
(llet-e)	$(\texttt{letrec}\ Env_1, x = (\texttt{letrec}\ Env_2\ \texttt{in}\ e_x)^S\ \texttt{in}\ e)$ $\to (\texttt{letrec}\ Env_1, Env_2, x = e_x\ \texttt{in}\ e)$
(lapp)	$C[((\texttt{letrec}\ Env\ \texttt{in}\ e_1)^S\ e_1)] \to C[(\texttt{letrec}\ Env\ \texttt{in}\ (e_1\ e_2))]$

Regeln des Markierungsalgorithmus:

$(\texttt{letrec}\ Env\ \texttt{in}\ e)^T$	$\to$	$(\texttt{letrec}\ Env\ \texttt{in}\ e^S)^V$
$(e_1\ e_2)^{S \vee T}$	$\to$	$(e_1^S\ e_2)^V$
$(\texttt{letrec}\ x = e, Env\ \texttt{in}\ C[x^S])$	$\to$	$(\texttt{letrec}\ x = e^S, Env\ \texttt{in}\ C[x^V])$ wenn e nicht markiert ist
$(\texttt{letrec}\ x = e_1, y = C[x^S], Env\ \texttt{in}\ e_2)$	$\to$	$(\texttt{letrec}\ x = e_1^S, y = C[x^V], Env\ \texttt{in}\ e_2)$ wenn e_1 nicht markiert ist und $C[x] \neq x$

Abbildung 5.1: Der Kalkül L_{need}

Applikative Simulation $\leq_{b,need}$ *ist definiert als der größte Fixpunkt von* F_{need}. Applikative Bisimulation $\sim_{b,need}$ *ist definiert als* $\sim_{b,need} := \leq_{b,need} \cap \geq_{b,need}$. *Die* offene Erweiterung $\sim^o_{b,need}$ *von* $\sim_{b,need}$ *als Relationen auf allen Ausdrücken ist definiert als* $e_1 \sim^o_{b,need} e_2$ *genau dann, wenn* $\sigma(e_1) \sim_{b,need} \sigma(e_2)$ *für alle Substitutionen* σ, *die Variablen durch geschlossene Ausdrücke ersetzen und für die* $\sigma(e_1)$ *und* $\sigma(e_2)$ *geschlossene Ausdrücke sind.*

Diese Definition enthält bereits als Vereinfachung, dass nur `letrec`-freie Abstraktionen und Ω als Argumente getestet werden müssen.

In (Schmidt-Schauß et al., 2010b) wurde das folgende Theorem bewiesen

Theorem 5.1.2. $\leq_{need} = \leq_{b,need}{}^o$

Der Beweis kann nicht durch Howes Methode (Howe, 1989, 1996) erbracht werden, da der Rahmen von Howe erfordert, dass die Operatoren der Sprache in kanonische und nicht-kanonische Partitionen geteilt werden können. Genau dann, wenn ein Ausdruck einen kanonischen Operator an oberster Position besitzt, ist der Ausdruck eine Antwort. Für das `letrec`-Konstrukt ist diese Einteilung jedoch nicht möglich, da ein Ausdruck $\texttt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \texttt{in}\ e$ je nach Form des Unterausdrucks e eine Antwort oder keine Antwort ist.

Daher wurde Theorem 5.1.2 in (Schmidt-Schauß et al., 2010b) auf einem Umweg bewiesen, dessen Beweisstruktur in Abbildung 5.2 dargestellt ist. Kontextuelle Gleichheit impliziert Bisi-

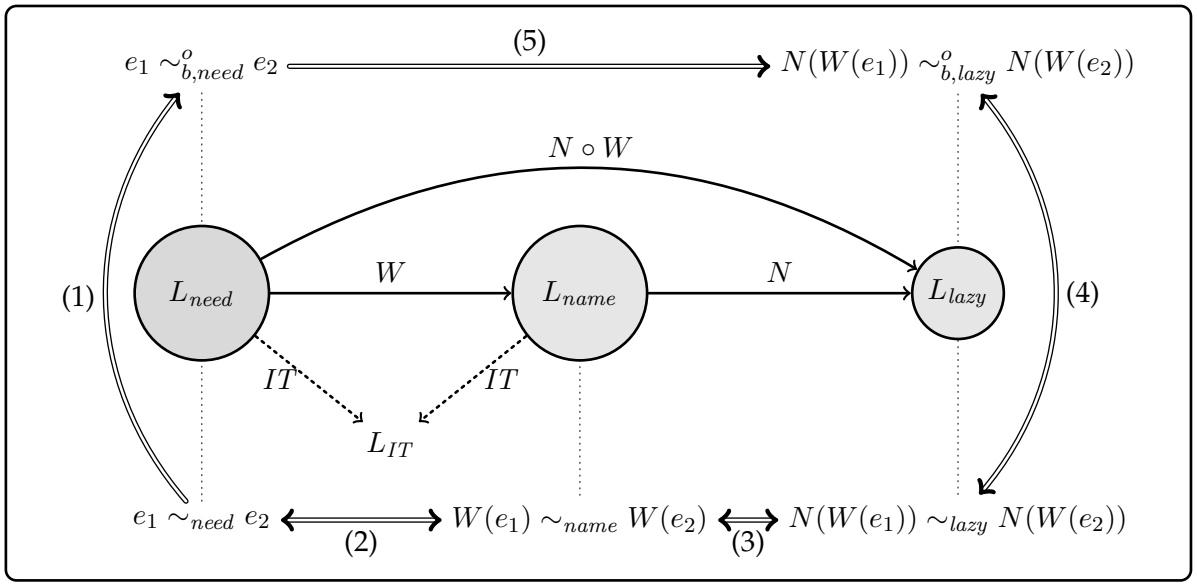


Abbildung 5.2: Beweisstruktur zur Identität von kontextueller Äquivalenz und Bisimulation in L_{need}

mulation in L_{need}, da $(\sim_{need})^c$ als F_{need}-dicht gezeigt werden kann[2], wobei $(\eta)^c$ die Einschränkung der Relation η auf geschlossene Ausdrücke ist. Für die offenen Relationen folgt diese Inklusion im Wesentlichen aus der Korrektheit der (lbeta)-Reduktion (dies entspricht Schritt (1) in Abbildung 5.2).

Für die Inklusion $\leq^o_{b,need} \subseteq \sim_{need}$ wurde zunächst der L_{need}-Kalkül mit der Übersetzung W in den Kalkül L_{name} übersetzt. Der L_{name}-Kalkül besitzt die selbe Syntax wie der L_{name}-Kalkül, verwendet jedoch anstelle der call-by-need Auswertung eine call-by-name Auswertung, die (β)-Reduktion und eine Kopierregel zum Kopieren beliebiger Ausdrücke enthält. Die Übersetzung W ist die Identität auf Ausdrücken und Kontexten. Die Übersetzung W ist voll-abstrakt[3] (dies entspricht der Äquivalenz (2) in Abbildung 5.2). Der Beweis verwendet die Übersetzung IT, die L_{need}-Ausdrücke (bzw. auch L_{name}-Ausdrücke) in *unendliche Bäume* L_{IT} übersetzt, indem alle `letrec`-Bindungen (unendlich oft) entfaltet werden und anschließend entfernt werden, d. h. sämtliche durch `letrec` gebundenen Variablen werden durch den entsprechenden unendlichen Baum ersetzt. Anschließend wird eine call-by-name Reduktion auf den unendlichen Bäumen definiert, die eine adaptierte β-Reduktion auf den Bäumen durchführt. Dabei werden einzelne Redexe reduziert, jedoch müssen u. U. unendlich viele Baumpositionen in einem Schritt geändert werden. Z. B. ist der unendliche Baum zum Ausdruck $(\lambda x.\texttt{letrec}\ y = (y\ x)\ \texttt{in}\ y)\ r$ der Baum der Form $(\lambda x.((\ldots\ x)\ x)\ldots x)\ IT(r)$. Die Baum-$\beta$-Reduktion führt zum unendlichen Baum $((\ldots\ IT(r))\ IT(r))\ldots IT(r))$ und muss daher unendlich viele x durch r ersetzen. Daher wird diese Reduktion ähnlich zu „infinite outside-

[2] Eine Relation η ist F_{need}-dicht gdw. $\eta \subseteq F_{need}(\eta)$. Da der größte Fixpunkt $\leq_{b,need}$ per Definition die Vereinigung aller F_{need}-dichten Relationen ist, folgt für F_{need}-dichtes η sofort $\eta \subseteq \leq_{b,need}$; siehe (Gordon, 1994).

[3] vgl. Definition 3.1.2

in developments“ (Barendregt, 1984; Kennaway et al., 1997) in Top-Down-Strategie von oben nach unten im Baum angewendet. Nach Definition der May-Konvergenz auf den unendlichen Bäumen, lässt sich zeigen (siehe (Schmidt-Schauß, 2007; Schmidt-Schauß et al., 2010b)), dass die Übersetzungen $IT : L_{need} \to L_{IT}$ und $IT : L_{name} \to L_{IT}$ konvergenzäquivalent sind[4], woraus die Konvergenzäquivalenz von W gefolgert werden kann. Da W die Identitätsfunktion darstellt, folgt daraus auch die volle Abstraktheit von W[5].

Schließlich wurde in Schritt (3) (siehe Abbildung 5.2) die Übersetzung $N : L_{name} \to L_{lazy}$ definiert, die den Programmkalkül L_{name} in den Lazy-Lambda-Kalkül übersetzt. Diese Übersetzung entfernt sämtliche `letrec`-Ausdrücke, indem sie diese durch Multifixpunkt-Kombinatoren (siehe z. B. (Goldberg, 2005)) ersetzt. In (Schmidt-Schauß et al., 2010b) wurde zunächst die Konvergenzäquivalenz von N bewiesen. Da N kompositional ist, konnte Satz 3.1.5 verwendet werden, um volle Abstraktheit von N zu zeigen. Im Lazy-Lambda-Kalkül fallen kontextuelle Äquivalenz und (die offene Erweiterung der) applikativen Bisimulation zusammen (Abramsky, 1990), was durch (4) in Abbildung 5.2 dargestellt ist.

Schließlich wurde gezeigt, dass applikative Bisimulation in L_{need} die applikative Bisimulation in L_{lazy} bzgl. der Übersetzung $N \circ W$ impliziert (Schritt (5) in Abbildung 5.2). Hierfür wurde im Wesentlichen die Konvergenzäquivalenz von $N \circ W$ und eine alternative Beschreibung der Bisimulation verwendet.

Insgesamt gilt $\leq_{need} = \leq^o_{b,need}$ entsprechend Implikation (1) in Abbildung 5.2 und $\leq^o_{b,need} \subseteq \leq_{need}$ über die Kette von Implikationen (5), (4), (3) und (2).

Ein weiteres Resultat, dass sich aus der Beweisstruktur ableiten lässt, ist die Isomorphie der Quotienten der Äquivalenzklassen (bzgl. kontextueller Äquivalenz) zwischen L_{need} und L_{lazy}:

Theorem 5.1.3. *Sei $id : L_{lazy} \to L_{need}$ die natürliche Einbettung von L_{lazy} in L_{need} und $\iota : \mathcal{E}_{lazy}/\sim_{lazy} \to \mathcal{E}_{need}/\sim_{need}$ mit $\iota([e]_{\sim_{lazy}}) := [id(e)]_{\sim_{need}}$. Dann ist ι ein Isomorphismus.*

Als weiteres Resultat wurde in (Schmidt-Schauß et al., 2010b) gezeigt, dass sämtliche Resultate auch auf den call-by-need Kalkül mit `letrec` aus (Ariola & Felleisen, 1997) übertragbar sind, der eine andere Standardreduktion als L_{need} verwendet.

Aktuelle Arbeiten (Schmidt-Schauß et al., 2012) zeigen, dass das gleiche Resultat auch für den syntaktisch reicheren call-by-need Kalkül mit letrec, case und Konstruktoren und Haskells seq-Operator (Schmidt-Schauß et al., 2008c) auf analoge Weise bewiesen werden kann.

5.2 Nichtextensionalität nichtdeterministischer letrec-Kalküle

In (Schmidt-Schauß et al., 2011) wurde ein erweiterter nichtdeterministischer call-by-need lambda Kalkül mit letrec untersucht. Dieser Kalkül namens L_S erweitert L_{need} zum einen um Datenkonstruktoren und case-Ausdrücke und zum anderen um erratische Auswahl (analog

[4] vgl. Definition 3.1.2
[5] Vgl. Satz 3.1.5.

zum Kalkül L_{lazynd} in Beispiel 2.1.8). Wir verzichten an dieser Stelle auf die genaue Angabe aller Komponenten von L_S, geben jedoch die Syntax der Ausdrücke an:

$$e, e_i \in \mathcal{E}_S ::= x \mid \lambda x.e \mid (e_1\ e_2) \mid \texttt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \texttt{in}\ e \mid (c\ e_1\ \ldots\ e_{\mathrm{ar}(c)})$$
$$\mid \texttt{case}_T\ e\ (c_1\ x_{1,1}\ \ldots\ x_{1,\mathrm{ar}(c_1)} \to e_1) \ldots (c_{|T|}\ x_{|T|,1}\ \ldots\ x_{|T|,\mathrm{ar}(c_{|T|})} \to e_{|T|}) \mid (e_1 \oplus e_2)$$

Der Kalkül besitzt Datenkonstruktoren c mit einer festen Stelligkeit $\mathrm{ar}(c)$, jeder Datenkonstruktor gehört zu genau einem Typkonstruktor T. Konstruktoranwendungen $(c\,e_1\ \ldots\ e_{\mathrm{ar}(c)})$ dürfen nur völlig gesättigt auftreten. Für jeden Typkonstruktor T gibt es ein $\texttt{case}_T$-Konstrukt, welches genau eine case-Alternative für jeden Datenkonstruktor des Typkonstruktors T enthält. Z. B. kann ein solcher Typkonstruktor der Typ $Bool$ mit den Datenkonstruktoren `True` und `False` sein. Der case-Ausdruck $(\texttt{case}_{Bool}\ e\ (\texttt{True} \to e_1)\ (\texttt{False}\ e_2))$ entspricht dann gerade dem if-then-else-Ausdruck $\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2$. Choice-Ausdrücke $(e_1 \oplus e_2)$ können entweder zu e_1 oder zu e_2 auswerten. Antworten sind alle WHNFs, wobei WHNFs Abstraktionen, Konstruktoranwendungen, wobei die Argumente Variablen sind, und Ausdrücke $\texttt{letrec}\ Env\ \texttt{in}\ v$, wobei v eine Abstraktion oder eine Konstruktoranwendung mit Variablen als Argumente ist, sind.

In (Schmidt-Schauß et al., 2011) werden die kontextuellen Präordnungen $\leq_{\downarrow}$, $\leq_{\downarrow,\Downarrow}$ und $\leq_{\downarrow,\Downarrow}$ bezüglich May-, Should- und Must-Konvergenz (siehe Abschnitt 2.2.2) und die entsprechenden kontextuellen Äquivalenzen untersucht.

Als wichtigste Eigenschaft wurde in (Schmidt-Schauß et al., 2011) die Extensionalität von Programmkalkülen betrachtet:

Definition 5.2.1. *Gegeben sei ein Programmkalkül D, der Applikation und Abstraktionen in seiner Syntax erlaubt. Sei $\sim$ eine Gleichheitsrelation auf D-Ausdrücken. Zwei geschlossene D-Ausdrücke e_1, e_2 sind* extensional gleich bzgl. $\sim$, *wenn für alle geschlossenen Ausdrücke r gilt: $(e_1\ r) \sim (e_2\ r)$. D ist* extensional bzgl. einer Menge von Ausdrücken Q und einer Gleichheitsrelation $\sim$, *wenn für alle Ausdrücke $e_1, e_2 \in Q$ gilt: e_1 und e_2 sind extensional gleich bzgl. $\sim$ gdw. $e_1 \sim e_2$ gilt.*

Es ist leicht zu zeigen, dass L_S nicht extensional bezüglich aller Ausdrücke ist. Ein Beispiel aus (Mann, 2005a) zeigt, dass L_S auch nicht extensional bezüglicher der Menge aller WHNFs ist. Hingegen ist der Kalkül L_{need} zwar nicht extensional bezüglich aller Ausdrücke (betrachte die Ausdrücke Ω und $\lambda x.\Omega$), aber extensional bezüglich der Menge der Antworten.

Das wesentliche Resultat aus (Schmidt-Schauß et al., 2011) ist, dass L_S nicht extensional bzgl. der Menge der Abstraktionen ist. Der folgende Satz liefert das Gegenbeispiel hierzu:

Satz 5.2.2. *Seien s_i, p_i und seqp definiert als:*

$$s_1 := \lambda x.(p_1 \oplus p_2) \qquad p_1 := (\texttt{False}, seqp\ x\ \texttt{False})$$
$$s_2 := \lambda x.(p_1 \oplus p_2 \oplus p_3) \qquad p_2 := seqp\ x\ (\texttt{False}, \texttt{True})$$
$$seqp := \lambda x.\texttt{case}_{Pair}\ x\ ((a, b) \to y) \qquad p_3 := (\texttt{False}, seqp\ x\ \texttt{True})$$

Es gilt:

- s_1 *und* s_2 *sind extensional bzgl.* $\sim_{\downarrow}$, $\sim_{\downarrow,\Downarrow}$ *und* $\sim_{\downarrow,\Downarrow}$.
- $s_1 \not\sim_{\downarrow} s_2$ *und daher auch* $s_1 \not\sim_{\downarrow,\Downarrow} s_2$ *und* $s_1 \not\sim_{\downarrow,\Downarrow} s_2$.

Der unterscheidende Kontext ist $C := \texttt{letrec}\ y = ([\cdot]\ y)\ \texttt{in}\ (\texttt{if}\ snd\ y\ \texttt{then True else}\ \Omega)$[6], wobei $snd := \lambda x.\texttt{case}_{Pair}\ x\ ((a,b) \to b)$. Dann kann $C[s_1]$ nicht konvergieren (d. h. $C[s_1]\Uparrow$), während $C[s_2]$ zu `True` auswerten kann und daher $C[s_2]\downarrow$ gilt.

Da s_1 und s_2 (geschlossene) Abstraktionen sind, folgt direkt:

Theorem 5.2.3. *L_S ist nicht extensional bezüglich der Menge der geschlossenen Abstraktionen und der Gleichheitsrelationen $\sim_{\downarrow}$, $\sim_{\downarrow,\Downarrow}$ und $\sim_{\downarrow,\Downarrow}$.*

Da die applikative Bisimulation (auch für nichtdeterministische Kalküle vgl. Definitionen 2.2.7, 2.2.8 und 5.1.1) gerade Ausdrücke zu Abstraktionen auswertet und anschließend die Anwendung der Abstraktionen auf Argumente durchführt, zeigt diese Nichtextensionalität, dass alle üblichen Definitionen der applikativen Bisimulation in L_S nicht korrekt sind, d. h. applikative Bisimulation impliziert die kontextuelle Gleichheit nicht.

In (Schmidt-Schauß et al., 2011) wurde des Weiteren gezeigt, dass das Gegenbeispiel auch auf nichtdeterministische Lambda-Kalküle mit call-by-value Auswertung übertragbar ist. Eine offene Frage ist, ob die Nichtextensionalität auch für den L_S-Kalkül ohne Datenstrukturen und `case`-Ausdrücke (was dem L_{need}-Kalkül erweitert um erratische Auswahl entspricht) gilt. Das Gegenbeispiel ist in diesem Fall nicht übertragbar.

5.3 Kontextuelle Gleichheit für polymorph getypte Programmkalküle

Der Kalkül LS (Schmidt-Schauß et al., 2008c) entspricht im Wesentlichen dem Kalkül L_S aus dem letzten Kapitel, wobei der Operator $\oplus$, der die erratische Auswahl darstellt, nicht vorhanden ist. Daher ist LS ein deterministischer Programmkalkül. Im Unterschied zu L_S verfügt LS über Haskells `seq`-Operator, der strikte Auswertung ermöglicht, da $\texttt{seq}\ a\ b$, informell gesagt, zunächst den Ausdruck a auswertet und im Anschluss als Resultat den Ausdruck b liefert. Der Kalkül LS wurde in (Schmidt-Schauß et al., 2008c) eingeführt und als ungetypte Kernsprache für Haskell verwendet, um die Korrektheit der Nöckerschen Striktheitsanalyse nachzuweisen. Dabei wurden viele Programmtransformationen als korrekt gezeigt.

Die Verwendung einer ungetypten Kernsprache kann dadurch gerechtfertigt werden, dass in getypten Sprachen mehr Gleichheiten gelten als in ungetypten Sprachen (eingeschränkt auf korrekt getypte Gleichheiten), da die kontextuelle Gleichheit in der ungetypten Sprache alle, aber in der getypten Sprache weniger (nur die korrekt getypten) Kontexte mit einbezieht. Zum anderen kann man argumentieren, dass während des Compilierens zunächst der Typcheck durchgeführt wird, die Optimierungen und Transformationen dann aber auf der ungetypten Sprache (nach Wegwerfen der Typinformation) durchgeführt werden.

Trotzdem oder auch gerade darum, stellen sich die Fragen, welche Gleichheiten ausschließlich in der getypten Sprache gelten, wie man diese Gleichheit definiert, sodass Programmtransformationen lokal auf Unterausdrücke angewendet werden können (ohne einen erneuten

[6] Hierbei ist der `if-then-else`-Ausdruck entsprechend als `case`-Ausdruck zu kodieren.

Typcheck des Programms durchzuführen) und schließlich wie solche Gleichheiten bewiesen werden können. Da Haskell ein Hindley-Milner Typsystem (Hindley, 1969; Milner, 1978; Damas & Milner, 1982) mit prädikativem Polymorphismus verwendet, ist ein solches Typsystem von besonderem Interesse.

In (Sabel et al., 2009) wurden genau diese Fragestellungen untersucht, indem eine polymorph getypte Variante des LS-Kalküls definiert und analysiert wurde.

Der wesentliche Kern der Resultate in (Sabel et al., 2009) ist die Verwendung von Typmarkierungen an allen Unterausdrücken eines Ausdrucks zusammen mit Bedingungen, welche die Wohl-Getyptheit bezüglich eines polymorphen, prädikativen Typsystems mit `let`-Polymorphismus eines Ausdrucks sicherstellen. Schließlich wird eine Standardreduktion definiert, die direkt auf den typmarkierten Ausdrücken arbeitet, die Wohl-Getyptheit durch Vererbung der Markierungen erhält und zudem äquivalent (bzgl. der May-Konvergenz) zur Reduktion ohne Typmarkierungen (wie sie bspw. in (Schmidt-Schauß et al., 2008c) verwendet wird) ist.

Entsprechend sind Programmtransformationen auch auf typmarkierten Ausdrücken definiert und Ausdrücke werden nur dann in andere Ausdrücke transformiert, wenn die oberste Typmarkierung übereinstimmt. Die Verwendung dieses Systems hat zum einen den Vorteil, dass die Anwendbarkeit von Programmtransformationen lokal am Unterausdruck entscheidbar ist, d. h. die Betrachtung des Gesamtprogramms ist hierfür nicht notwendig. Ein weiterer Vorteil liegt darin, dass die Diagrammmethode zum Korrektheitsnachweis von Programmtransformationen (siehe Abschnitt 4.2) für getypte Programmtransformationen adaptiert und daher weiterhin verwendet werden konnte.

Bezüglich der getypten Variante des Kalküls LS konnte in (Sabel et al., 2009) zum einen nachgewiesen werden, dass die bekannten Programmgleichheiten aus (Schmidt-Schauß et al., 2008c) auf den getypten Kalkül übertragen werden können, und zum anderen konnte die Korrektheit von getypten Programmtransformationen, die ausschließlich im getypten Kalkül gelten, nachgewiesen werden, indem die Diagrammmethode verwendet wurde. Eine solche korrekte Programmtransformation ist:

$$(\texttt{case}_{List}\ e\ (\texttt{Nil} \to \texttt{Nil})\ (\texttt{Cons}\ x\ xs) \to (\texttt{Cons}\ x\ xs)) : [T] \sim_{\downarrow} e : [T] \text{ für alle Typen } T.$$

Sie besagt, dass der `case`-Ausdruck, der e zerlegt, aber wieder wie vorher zusammensetzt, entfernt werden kann. Diese Transformation gilt nicht im ungetypten Kalkül, da z. B. für $e = \texttt{True}$ gilt: $e{\downarrow}$ aber $\texttt{case}_{List}\ e\ \ldots{\Uparrow}$.

Als Fazit lässt sich ziehen, dass die Arbeiten in (Sabel et al., 2009) zeigen, dass auch kontextuelle Gleichheiten für den polymorph getypten Fall nachweisbar sind, und dass bestehende Techniken und Methoden auf diesen Fall übertragen werden können. Dabei ist die Markierung aller Unterausdrücke mit Typen ein wichtiges Mittel, um die Anwendbarkeit von Programmtransformation lokal entscheiden zu können.

6

Semantik von nebenläufigen Programmiersprachen am Beispiel Concurrent Haskell mit Futures

In diesem Kapitel erläutern wir die Ergebnisse der Untersuchungen zum *CHF*-Kalkül, der ein Modell für Concurrent Haskell erweitert um Futures ist. Das Kapitel fasst daher die drei Arbeiten (Sabel & Schmidt-Schauß, 2011b), (Sabel, 2012b) und (Sabel & Schmidt-Schauß, 2012) zusammen.

In (Sabel & Schmidt-Schauß, 2011b) wurde der *CHF*-Kalkül eingeführt und motiviert. Aufbauend auf der kontextuellen Semantik mit May- und Should-Konvergenz wurde eine Reihe von Programmtransformationen als korrekt nachgewiesen. Zudem wurde dort gezeigt, dass die verzögerte call-by-need Auswertung äquivalent zur voll kopierenden call-by-name Auswertung ist.

In (Sabel & Schmidt-Schauß, 2012) wurde die Erweiterung des rein funktionalen Kerns von Haskell hin zum *CHF*-Kalkül untersucht. Der *CHF*-Kalkül erweitert die funktionale Kernsprache von Haskell um monadische Ein- und Ausgabe, um Nebenläufigkeit mit sogenannten MVars als Synchronisationskonstrukte und um Futures. In (Sabel & Schmidt-Schauß, 2012) konnte gezeigt werden, dass diese Erweiterung *konservativ* ist, d. h. sämtliche Gleichheiten aus der funktionalen Kernsprache gelten weiterhin im *CHF*-Kalkül. Auch die Grenzen der Konservativität wurden untersucht, indem gezeigt wurde, dass die Erweiterung des *CHF*-Kalküls um sogenannte lazy Futures nicht konservativ bezüglich der funktionalen Kernsprache ist. Da sowohl Futures als auch lazy Futures mithilfe der Bibliotheksfunktion `unsafeInterleaveIO` in Concurrent Haskell implementierbar sind, zeigt dies eine semantische Grenze zur Verwendung dieses Konstrukts auf.

Schließlich wurde in (Sabel, 2012b) eine alternative operationale Semantik für den *CHF*-Kalkül in Form einer abstrakten Maschine definiert, die dabei in mehreren Schritten aus Se-

stofts abstrakter Maschine für den puren call-by-need Lambda Kalkül (Sestoft, 1997) abgeleitet wurde. Als Resultat wurde gezeigt, dass die erhaltene Maschine ein korrekter Evaluator für den *CHF*-Kalkül ist, da sie konvergenzäquivalent zur ursprünglichen in (Sabel & Schmidt-Schauß, 2011b) definierten Reduktionssemantik ist.

6.1 Concurrent Haskell

Concurrent Haskell (Peyton Jones et al., 1996; Peyton Jones, 2001; Peyton Jones & Singh, 2009) erweitert die funktionale Programmiersprache Haskell um nebenläufige Threads und sogenannte MVars innerhalb der `IO`-Monade (Peyton Jones & Wadler, 1993; Wadler, 1995; Peyton Jones, 2001). Die Primitive `forkIO :: IO a -> IO ThreadId` erzeugt einen nebenläufigen Thread und liefert als Ergebnis einen Identifier für den Thread. MVars sind synchronisierende Speicherplätze die leer oder gefüllt sein können. Mit `newMVar :: a -> IO (MVar a)` wird eine gefüllte MVar erzeugt. Die Primitiven `takeMVar :: MVar a -> IO a` und `putMVar :: MVar a -> a -> IO ()` leeren bzw. füllen den Speicherplatz und führen zum Blockieren, falls der Speicherplatz bereits leer beziehungsweise schon gefüllt ist.

Nebenläufige Futures (Baker & Hewitt, 1977; Halstead, 1985), wie wir sie bereits im imperativen call-by-value Lambda-Kalkül in Abschnitt 3.2.2 gesehen haben, sind Variablen deren Werte durch nebenläufige Threads ermittelt werden (in der Zukunft). Die Variablen können aber wie Referenzen schon durch andere Threads verwendet werden, solange der Wert der Future nicht gebraucht wird. Wird der Wert einer Future benötigt, so blockiert der entsprechende Thread solange, bis der Wert verfügbar ist. Daher kann mithilfe von Futures die Synchronisation alleinig durch Datenabhängigkeiten programmiert werden, was einen deklarativen Programmierstil für Nebenläufigkeit erlaubt.

Man unterscheidet zwischen expliziten und impliziten Futures. Während bei expliziten Futures der Wert einer Future durch ein explizites Kommando (oft `force` genannt) gelesen werden muss, geschieht dies bei impliziten Futures automatisch, da sie wie Variablen behandelt werden. Implizite Futures erlauben daher einen wesentlich deklarativeren Programmierstil, da der Programmierer nicht wissen muss, wann genau der Wert einer Future benötigt wird.

Explizite Futures sind im Allgemeinen einfach aus bestehenden Konstrukten der nebenläufigen Programmierung als Bibliothek zu implementieren, während implizite Futures oft eine Spracherweiterung erfordern.

In Concurrent Haskell können explizite Futures durch MVars dargestellt werden und die Operationen `efuture` zum Erzeugen einer Future sowie `force` zum expliziten Anfordern einer Future können mit den vorhanden Primitiven implementiert werden:

```
type EFuture a = MVar a

efuture :: IO a -> EFuture a
efuture act = do
   fut <- newEmptyMVar
   forkIO (act >>= putMVar fut)
```

```
    return fut

force :: EFuture a -> IO a
force fut =  do
  res <- takeMVar fut
  putMVar fut res
  return res
```

Tatsächlich können auch implizite Futures in Concurrent Haskell unter Zuhilfenahme der Operation `unsafeInterleaveIO` implementiert werden, wobei `unsafeInterleaveIO` nicht zum Sprachumfang von Haskell gehört, allerdings in nahezu allen Implementierungen von Haskell wie beispielsweise dem GHC vorhanden ist. Der Operator `unsafeInterleaveIO` verzögert eine monadische Operation in der `IO`-Monade und bricht die strikte Sequentialisierung der Monade auf (daher wird die Primitive als nicht sicher (= unsafe)) bezeichnet. Implizite Futures können implementiert werden durch:

```
future :: IO a -> IO a
future act = do
   fut <- newEmptyMVar
   forkIO (act >>= putMVar fut)
   unsafeInterleaveIO (takeMVar fut)
```

Im Grunde führt die `future`-Funktion die beiden Operationen `efuture` und `force` direkt nacheinander aus, aber das Entnehmen des Werts mittels `takeMVar` wird durch `unsafeInterleaveIO` solange verzögert, bis der Wert wirklich benötigt wird.

6.2 Der CHF-Kalkül

Die Untersuchungen in (Sabel & Schmidt-Schauß, 2011b) haben diese Erweiterung von Concurrent Haskell als Motivation. Daher wurde der *CHF*-Kalkül als Modell dieser Sprache in (Sabel & Schmidt-Schauß, 2011b) entworfen und untersucht. Teilweise ähnelt der Kalkül dem call-by-value Lambda-Kalkül mit Futures $\lambda(\mathsf{fh})$ (see Abschnitt 3.2.2), allerdings gibt es einige Unterschiede: Die Auswertung in *CHF* folgt der call-by-need Strategie, Seiteneffekte sind nur innerhalb der IO-Monade zulässig und erfolgreiche Prozesse sind anders definiert: Während in $\lambda(\mathsf{fh})$ sämtliche Threads terminieren müssen, bevor der Gesamtprozess erfolgreich ist, gibt es in *CHF* einen ausgezeichneten Thread (den Main-Thread), dessen Terminierung auch zur Terminierung des Gesamtprozesses führt. Dies bildet das Verhalten von Concurrent Haskell ab: Sobald der Main-Thread dort terminiert, werden alle nebenläufigen Threads abgebrochen.

Wir erläutern zunächst die Syntax von *CHF*. Diese ist zweistufig: Auf der oberen Ebene befinden sich *Prozesse* und einige andere Komponenten, und innerhalb von Prozessen sind in der unteren Stufe *Ausdrücke* eines erweiterten Lambda-Kalküls zu finden.

Die (ungetypte) Syntax von Prozessen $P \in \mathit{Proc}$ ist durch die folgende Grammatik definiert, wobei x eine Variable bezeichnet und e ein Ausdruck ist.

$$P, P_i \in \mathit{Proc} ::= P_1 \mid P_2 \mid \nu x P \mid x \Leftarrow e \mid x = e \mid x\,\mathsf{m}\,e \mid x\,\mathsf{m}\,-$$

Dabei steht $P_1 \mid P_2$ für die parallele Komposition und $\nu x P$ schränkt den Gültigkeitsbereich der Variablen x auf den Prozess P ein. $x \Leftarrow e$ stellt eine nebenläufige Future (einen Thread) dar, die den Ausdruck e auswertet und das Resultat an die Variable x bindet. $x = e$ ist eine globale Bindung, dabei ist die Variable x ist an den Ausdruck e gebunden. Solche Bindungen dürfen rekursiv sein. $x\,\mathsf{m}\,e$ und $x\,\mathsf{m}\,-$ stellen volle bzw. leere MVars dar. $x\,\mathsf{m}\,e$ ist die MVar namens x, die den Inhalt e enthält. $x\,\mathsf{m}\,-$ ist die leere MVar namens x. Als syntaktische Bedingung wird hinzugefügt, dass es in einem Prozess maximal einen ausgezeichneten Thread gibt, den *Main-Thread*. Er wird durch $x \overset{\mathsf{main}}{\Longleftarrow} e$ dargestellt.

(Ungetypte) Ausdrücke $e \in \mathit{Expr}$ werden durch die folgende Grammatik gebildet, wobei eine Untermenge der Ausdrücke, die *monadischen Ausdrücke* $me \in \mathit{MExpr}$ darstellen.

$$\begin{aligned}
e, e_i \in \mathit{Expr} ::= {} & x \mid me \mid \lambda x.e \mid (e_1\ e_2) \mid c\ e_1 \ldots e_{\mathrm{ar}(c)} \mid \texttt{seq}\ e_1\ e_2 \\
& \mid \texttt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \texttt{in}\ e \\
& \mid \texttt{case}_T\ e\ \texttt{of}\ (c_{T,1}\ x_1 \ldots x_{\mathrm{ar}(c_{T,1})} \to e_1) \ldots\ (c_{T,|T|}\ x_1 \ldots x_{\mathrm{ar}(c_{T,|T|})} \to e_{|T|}) \\
me \in \mathit{MExpr} ::= {} & \texttt{return}\ e \mid e_1\ \texttt{>>=}\ e_2 \mid \texttt{future}\ e \\
& \mid \texttt{takeMVar}\ e \mid \texttt{newMVar}\ e \mid \texttt{putMVar}\ e_1\ e_2
\end{aligned}$$

Wie der Lambda-Kalkül beinhalten Ausdrücke Variablen x, Abstraktionen $\lambda x.e$ und Anwendungen $(e_1\ e_2)$. Darüber hinaus gibt es (voll gesättigte) Konstruktoranwendungen $(c\ e_1 \ldots e_{\mathrm{ar}(c)})$, `case`-Ausdrücke, `seq`-Ausdrücke `seq` $e_1\ e_2$ zur sequentiellen Auswertung und `letrec`-Ausdrücke, die es erlauben lokale Bindungen zu konstruieren.

Ausdrücke umfassen zusätzlich die Menge der monadischen Ausdrücke: (`return` e) verpackt einen beliebigen Ausdruck als monadische Aktion, der binäre `>>=`-Operator komponiert zwei monadische Aktionen zu einer Sequenz. Die Ausdrücke (`newMVar` e), (`takeMVar` e) und (`putMVar` $e_1\ e_2$) dienen dem Erzeugen und dem Zugriff auf MVars (wie in Concurrent Haskell). Schließlich gibt es noch den Operator `future` e zum Erzeugen einer Future, welche die Aktion e ausführt.

Für Ausdrücke und Prozesse führen die Konstrukte $\nu x.P$, $\lambda x.e$, `letrec` $x_1 = e_1, \ldots, x_n = e_n$ `in` e, und `case`-Alternativen Bindungsbereiche von Variablen ein, was die offensichtlichen Notationen von gebundenen und freien Variablen, sowie die α-Umbenennung und α-Äquivalenz induziert.

Eingeführte Variablen eines Prozesses sind alle Namen der Threads (die Futures), die Namen der MVars und die linken Seiten der globalen Bindungen. Ein Prozess P ist genau dann *wohlgeformt*, wenn alle eingeführten Variablen paarweise verschieden sind und P maximal einen Main-Thread besitzt.

Für Prozesse wird (ähnlich zum π-Kalkül) eine strukturelle Kongruenz verwendet, die in Abbildung 6.1a definiert ist, und das Umsortieren von Komponenten und Bewegen von ν-Bindern innerhalb eines Prozesses erlaubt.

Der *CHF*-Kalkül ist monomorph typisiert (wobei Datenkonstruktoren überladen verwendet werden (und daher eher polymorph sind)). Dies ist eine Einschränkung gegenüber Concurrent Haskell, da dort *polymorphe* Typen vorhanden sind. Die Syntax von Typen $\tau \in \mathit{Type}$ ist durch die folgende Grammatik definiert:

$$\tau, \tau_i \in \mathit{Type} ::= \texttt{IO}\ \tau \mid (T\ \tau_1 \ldots \tau_{\mathrm{ar}(T)}) \mid \texttt{MVar}\ \tau \mid \tau_1 \to \tau_2$$

Hierbei ist T ein Typkonstruktor, wobei jeder Typkonstruktor eine Stelligkeit $\mathrm{ar}(T) \in \mathbb{N}_0$ besitzt, und innerhalb von Typen nur vollständig gesättigte Typkonstruktoranwendungen $(T\ \tau_1 \ldots \tau_{\mathrm{ar}(T)})$ auftreten dürfen. `IO` τ ist der Typ einer `IO`-Aktion mit Rückgabewert vom Typ τ, `MVar` τ ist der Typ einer MVar mit Inhaltstyp τ und $\tau_1 \to \tau_2$ stellt einen Funktionstyp dar.

Die vollständigen Typregeln sind in (Sabel & Schmidt-Schauß, 2011b) zu finden, erwähnenswert ist zum einen die Typisierung von Futures: Eine Future $x \Leftarrow e$ ist wohl-getypt, wenn e vom Typ `IO` τ und x vom Typ τ ist. Zum anderen ist die Typisierung von `seq`-Ausdrücken eingeschränkter als in Haskell: In `seq` $e_1\ e_2$ darf e_1 keinen `IO`-Typ oder `MVar`-Typ besitzen. Der Grund liegt darin, dass anderenfalls die monadischen Gesetze für die `IO`-Monade nicht gelten (dies gilt auch für Haskell).

6.2.1 Operationale Semantik

Die operationale Semantik von *CHF* ist in (Sabel & Schmidt-Schauß, 2011b) als Small-Step-Reduktion auf Prozessen definiert. Dabei wird die verzögerte Auswertung berücksichtigt, d. h. nur notwendige Ausdrücke werden ausgewertet und Sharing wird verwendet, um Doppelauswertungen zu vermeiden. In Abbildung 6.1b werden verschiedene Klassen von Kontexten festgelegt, um schließlich die Standardreduktion $\xrightarrow{CHF}$ des *CHF*-Kalküls zu definieren (Abbildung 6.1c). Die Standardreduktion umfasst dabei zum einen Regeln für die Auswertung monadischer Operatoren und zum anderen Regeln für die Auswertung von funktionalen Ausdrücken. Wir erläutern das Zusammenspiel der Regeln (fork) und (unIO): Die Reduktionsregel (fork) wertet eine `future`-Operation aus, indem eine neue Future erzeugt wird. Die Berechnung der Future ist erfolgreich beendet, wenn sie von der Form `return` e ist. In diesem Fall wird die Regel (unIO) angewendet, um die Future zu entfernen. Das Ergebnis wird dabei als normale Bindung aufgehoben.

Für die Auswertung der funktionalen Ausdrücke ist anzumerken, dass monadische Operatoren innerhalb von funktionalen Ausdrücken genau wie Datenkonstruktoren behandelt werden.

CHF ist offensichtlich ein nichtdeterministischer Programmkalkül, da zu jedem Schritt ein Thread (samt evtl. weiterer Komponenten) ausgewählt wird (unter allen Threads), der einen Schritt in der Berechnung machen darf.

Strukturelle Kongruenz:
$\equiv$ ist die kleineste Kongruenz auf Prozessen, die die folgenden Regeln erfüllt:

$$\begin{aligned} P_1 \mid P_2 &\equiv P_2 \mid P_1 \\ P_1 \mid (P_2 \mid P_3) &\equiv (P_1 \mid P_2) \mid P_3 \\ (\nu x.P_1) \mid P_2 &\equiv \nu x.(P_1 \mid P_2), \text{ falls } x \notin FV(P_2) \\ \nu x_1.\nu x_2.P &\equiv \nu x_2.\nu x_1.P \\ P_1 &\equiv P_2, \text{ falls } P_1 \text{ und } P_2 \ \alpha\text{-äquivalente Prozesse sind } (P_1 =_\alpha P_2) \end{aligned}$$

(a) Strukturelle Kongruenz

$\mathbb{D} \in \mathit{PC} ::= [\cdot] \mid \mathbb{D} \mid P \mid P \mid \mathbb{D} \mid \nu x.\mathbb{D}$ $\qquad$ $\mathbb{M} \in \mathit{MC} ::= [\cdot] \mid \mathbb{M}\ \texttt{>>=}\ e$

$\mathbb{E} \in \mathit{EC} ::= [\cdot] \mid (\mathbb{E}\ e) \mid (\texttt{case}\ \mathbb{E}\ \texttt{of}\ \mathit{alts}) \mid (\texttt{seq}\ \mathbb{E}\ e)$

$\mathbb{F} \in \mathit{FC} ::= \mathbb{E} \mid (\texttt{takeMVar}\ \mathbb{E}) \mid (\texttt{putMVar}\ \mathbb{E}\ e)$

$\mathbb{L} \in \mathit{LC} ::= x \Leftarrow \mathbb{M}[\mathbb{F}] \mid x \Leftarrow \mathbb{M}[\mathbb{F}[x_n]] \mid x_n = \mathbb{E}_n[x_{n-1}] \mid \ldots \mid x_2 = \mathbb{E}_2[x_1] \mid x_1 = \mathbb{E}_1$
wobei $\mathbb{E}_2, \ldots \mathbb{E}_n$ nicht der leere Kontext sind.

$\widehat{\mathbb{L}} \in \widehat{\mathit{LC}} ::= x \Leftarrow \mathbb{M}[\mathbb{F}] \mid x \Leftarrow \mathbb{M}[\mathbb{F}[x_n]] \mid x_n = \mathbb{E}_n[x_{n-1}] \mid \ldots \mid x_2 = \mathbb{E}_2[x_1] \mid x_1 = \mathbb{E}_1$
wobei $\mathbb{E}_1, \mathbb{E}_2, \ldots \mathbb{E}_n$ nicht der leere Kontext sind.

(b) Prozess-, Monadische, Evaluations- und Forcing-Kontexte

Monadische Berechnungen:

(lunit) $y \Leftarrow \mathbb{M}[\texttt{return}\ e_1\ \texttt{>>=}\ e_2] \xrightarrow{\mathit{CHF}} y \Leftarrow \mathbb{M}[e_2\ e_1]$

(tmvar) $y \Leftarrow \mathbb{M}[\texttt{takeMVar}\ x] \mid x\ \mathsf{m}\ e \xrightarrow{\mathit{CHF}} y \Leftarrow \mathbb{M}[\texttt{return}\ e] \mid x\ \mathsf{m}\ -$

(pmvar) $y \Leftarrow \mathbb{M}[\texttt{putMVar}\ x\ e] \mid x\ \mathsf{m}\ - \xrightarrow{\mathit{CHF}} y \Leftarrow \mathbb{M}[\texttt{return}\ ()] \mid x\ \mathsf{m}\ e$

(nmvar) $y \Leftarrow \mathbb{M}[\texttt{newMVar}\ e] \xrightarrow{\mathit{CHF}} \nu x.(y \Leftarrow \mathbb{M}[\texttt{return}\ x] \mid x\ \mathsf{m}\ e)$

(fork) $y \Leftarrow \mathbb{M}[\texttt{future}\ e] \xrightarrow{\mathit{CHF}} \nu z.(y \Leftarrow \mathbb{M}[\texttt{return}\ z] \mid z \Leftarrow e)$
wobei z frisch ist und der erzeugte Thread ist kein Main-Thread

(unIO) $y \Leftarrow \texttt{return}\ e \xrightarrow{\mathit{CHF}} y = e$ wenn der Thread nicht der Main-Thread ist

Funktionale Auswertung:

(cp) $\widehat{\mathbb{L}}[x] \mid x = v \xrightarrow{\mathit{CHF}} \widehat{\mathbb{L}}[v] \mid x = v$ wobei v eine Abstraktion oder eine Variable ist

(cpcx) $\widehat{\mathbb{L}}[x] \mid x = c\ e_1 \ldots e_n$ wenn c ein Konstruktor oder ein monadischer Operator ist
$\xrightarrow{\mathit{CHF}} \nu y_1, \ldots y_n.(\widehat{\mathbb{L}}[c\ y_1 \ldots y_n] \mid x = c\ y_1 \ldots y_n \mid y_1 = e_1 \mid \ldots \mid y_n = e_n)$

(mkbinds) $\mathbb{L}[\texttt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \texttt{in}\ e]$
$\xrightarrow{\mathit{CHF}} \nu x_1, \ldots, x_n.(\mathbb{L}[e] \mid x_1 = e_1 \mid \ldots \mid x_n = e_n)$

(lbeta) $\mathbb{L}[((\lambda x.e_1)\ e_2)] \xrightarrow{\mathit{CHF}} \nu x.(\mathbb{L}[e_1] \mid x = e_2)$

(case) $\mathbb{L}[\texttt{case}_T\ (c\ e_1\ \ldots\ e_n)\ \texttt{of}\ \ldots (c\ y_1\ \ldots\ y_n \to e) \ldots]$
$\xrightarrow{\mathit{CHF}} \nu y_1, \ldots, y_n.(\mathbb{L}[e] \mid y_1 = e_1 \mid \ldots \mid y_n = e_n])$

(seq) $\mathbb{L}[(\texttt{seq}\ v\ e)] \xrightarrow{\mathit{CHF}} \mathbb{L}[e]$ wenn v ein funktionaler Wert ist

Abschluss: Wenn $P_1 \equiv \mathbb{D}[P_1']$, $P_2 \equiv \mathbb{D}[P_2']$ und $P_1' \xrightarrow{\mathit{CHF}} P_2'$ dann $P_1 \xrightarrow{\mathit{CHF}} P_2$

(c) Standardreduktion

Abbildung 6.1: Operationale Semantik von CHF

Ein Prozess ist erfolgreich (und daher eine Antwort im Sinne eines getypten Programmkalküls, Definition 2.1.1), wenn er wohlgeformt ist, und der Main-Thread von der Form $x \overset{\mathsf{main}}{\Longleftarrow} \texttt{return}\ e$ ist.

Die kontextuelle Präordnung $\leq_{CHF}$ und Gleichheit $\sim_{CHF}$ für *CHF* sind definiert auf Basis der May- und Should-Konvergenz (siehe Abschnitt 2.2.2) bezüglich der Standardreduktion $\overset{CHF}{\longrightarrow}$, wobei für Prozesse sämtliche Prozesskontexte betrachtet werden und für Ausdrücke sämtliche Kontexte, die Prozesse sind und ihr Loch an Ausdrucksposition haben. Da *CHF* getypt ist, werden nur wohltypisierte Kontexte betrachtet und nur gleich getypte Ausdrücke werden von kontextuellen Gleichheit erfasst.

Ein wichtiges Resultat dabei ist, dass diese kontextuelle Gleichheit identisch zu einer Gleichheit ist, welche die May- und Should-Konvergenzprädikate nur für *faire* Reduktionsfolgen definiert. Fair bedeutet hierbei, dass jeder Thread nach endlich vielen Reduktionsschritten reduziert wird (falls er reduzibel ist). Im Grunde will man nur faire Reduktionsfolgen betrachten, da der Scheduler üblicherweise auf Fairness achtet. Die explizite Betrachtung macht die entsprechenden Beweise wesentlich aufwändiger. Da die kontextuelle Gleichheit ohne Beachtung der Fairness jedoch mit der kontextuellen Gleichheit mit Beachtung der Fairness zusammenfällt, ist es nicht notwendig diese explizit zu betrachten.

In (Sabel & Schmidt-Schauß, 2011b) konnte ein Kontextlemma in ähnlicher Weise wie in Abschnitt 4.1 für den *CHF*-Kalkül nachgewiesen werden. Für die Gleichheit von Ausdrücken reicht es daher, die May- und Should-Konvergenz in $D[F]$-Kontexten (siehe Abbildung 6.1b) zu betrachten.

6.3 Korrektheit von Programmtransformationen

In (Sabel & Schmidt-Schauß, 2011b) wurde die Korrektheit von einigen Transformationen im *CHF*-Kalkül nachgewiesen:

Satz 6.3.1. *Die Reduktionen (lunit), (nmvar), (fork), (unIO), (cp), (cpcx), (mkbinds), (lbeta), (case) und (seq) sind korrekte Programmtransformationen. Ferner sind (cp), (cpcx), (mkbinds), (lbeta), (case) und (seq) auch korrekt, wenn man die entsprechende Transformation auf beliebige Kontexte (anstelle der LC- und $\widehat{LC}$-Kontexte) erweitert.*

In (Sabel & Schmidt-Schauß, 2011b) wurde des Weiteren gezeigt, dass das Entfernen von unbenutzten Bindungen und MVars (d. h. Garbage Collection) eine korrekte Programmtransformation ist. Ebenso wurde gezeigt, dass eine generelle Kopierregel (gcp) der Form $x = e \mid C[x] \to x = e \mid C[e]$ korrekt ist.

Während für einige Korrektheitsnachweise die Diagrammmethode (siehe Abschnitt 4.2) verwendet werden konnte, funktionierte diese Methode für die Korrektheit der Kopierregeln (cp), (cpcx) und (gcp) nicht.

Daher wurde die bereits im `letrec`-Kalkül L_{need} (siehe Abschnitt 5.1) verwendete „infinite trees“-Methode für den nebenläufigen Prozesskalkül *CHF* in (Sabel & Schmidt-Schauß, 2011b) adaptiert. Dabei werden *CHF*-Prozesse und *CHF*-Ausdrücke durch eine Übersetzung

IT auf den Kalkül *CHFI* mit unendlichen Bäume abgebildet. *CHFI* ist analog zur *CHF*-Syntax von definiert: Prozesse enthalten jedoch keine Bindungen, Ausdrücke enthalten kein `letrec`-Konstrukt und die Grammatik für Ausdrücke wird coinduktiv definiert, d. h. sie kann unendliche große Ausdrücke enthalten. Die Übersetzung $IT : \mathit{CHF} \to \mathit{CHFI}$ entfaltet alle (globalen und lokalen) rekursive Bindungen $x = e$ unendlich oft und setzt sie direkt an die referenzierten Stellen ein, d. h. in den Bäumen kommen keine Variablen für Bindungen vor, an deren Stelle tritt der entsprechende unendliche Baum. Die Bindungen als Prozesskomponenten werden entfernt und Variablen, die in Bindungen der Form $x = x$ (oder auch über Ketten $x_1 = x_2, \ldots, x_n = x_{n-1}, x_{n+1} = x_1$) vorkommen, werden durch eine stets divergierende Konstante $\bot$ ersetzt.

Anschließend wurde für *CHFI* eine call-by-name Baum-Reduktion im Stil von „infinite outside-in developments“ (Barendregt, 1984; Kennaway et al., 1997) definiert. Nach Definition von May- und Should-Konvergenz auf unendlichen Bäumen wurde gezeigt, dass für alle Prozesse P gilt $P{\downarrow} \iff IT(P){\downarrow}$ und $P{\Downarrow} \iff IT(P){\Downarrow}$, d. h. die Übersetzung IT ist konvergenzäquivalent. Da die Kopierregeln den entsprechenden unendlichen Baum unverändert lassen, lässt sich die Korrektheit dieser Transformationen schließlich einfach folgern.

Die Technik der unendlichen Bäume wurde zudem dazu benutzt, nachzuweisen, dass eine call-by-name Reduktionsstrategie äquivalent (bezüglich May- und Should-Konvergenz) zur definierten Standardreduktion ist. Die call-by-name Auswertung kopiert dabei ganze Ausdrücke, verwendet die übliche (β)-Reduktion und erzeugt keine Bindungen zum Sharen von Unterausdrücken.

Schließlich wurde in (Sabel & Schmidt-Schauß, 2011b) gezeigt, dass die monadischen Gesetze der `IO`-Monade für *CHF* gelten:

Satz 6.3.2.

Für alle Typen τ_1, τ_2 und alle Ausdrücke $e_1 :: \tau_1$ und $e_2 :: \tau_1 \to \texttt{IO}\ \tau_2$ gilt:

$$\texttt{return}\ e_1\ \texttt{>>=}\ e_2 \sim_{\mathit{CHF}} e_2\ e_1.$$

Für alle Typen τ und alle Ausdrücke $e_1 :: \texttt{IO}\ \tau$ gilt:

$$e_1\ \texttt{>>=}\ \lambda x.\texttt{return}\ x \sim_{\mathit{CHF}} e_1.$$

Für alle Typen τ_1, τ_2, τ_3 und alle Ausdrücke $e_1 :: \texttt{IO}\ \tau_1, e_2 :: \tau_1 \to \texttt{IO}\ \tau_2, e_3 :: \tau_2 \to \texttt{IO}\ \tau_3$ gilt:

$$e_1\ \texttt{>>=}\ (\lambda x.(e_2\ x\ \texttt{>>=}\ e_3)) \sim_{\mathit{CHF}} (e_1\ \texttt{>>=}\ e_2)\ \texttt{>>=}\ e_3.$$

6.4 Konservativität von CHF

Die Untersuchungen in (Sabel & Schmidt-Schauß, 2011b) ließen die wesentliche Frage bezüglich Concurrent Haskell und auch bezüglich *CHF* offen, ob die, durch die IO-Operationen, durch die Nebenläufigkeit und durch die Futures zur puren Kernsprache hinzugefügte, Ausdruckskraft auch Auswirkungen auf die Gleichheiten der wohluntersuchten Kernsprache (einem erweiterten call-by-need Lambda-Kalkül mit letrec) als Einbettung in *CHF* hat. Genauer

gilt in (purem) Haskell das Prinzip der referentiellen Transparenz, d. h. der Wert einer Anwendung einer Funktion auf ein Argument ist stets eindeutig und hat keinerlei Seiteneffekte. Daher stellte sich die Frage, ob dieses Prinzip durch Hinzufügen der neuen Konstrukte verletzt wird. Um diesen Begriff formaler zu fassen, wurde in (Sabel & Schmidt-Schauß, 2012) die Fragestellung untersucht, ob alle Gleichungen der rein funktionalen Kernsprache auch in *CHF* weiterhin gültig sind, d. h. es wurde untersucht, ob *CHF* eine *konservative Erweiterung* des deterministischen rein funktionalen call-by-need Lambda-Kalküls mit letrec, seq, case und Konstruktoren ist. Ein positives Resultat impliziert, dass gleiche Funktionen auch durch die erweiterte Syntax nicht unterschieden werden können und sich daher alle (puren) Funktionen weiterhin so verhalten, als ob es keine Seiteneffekte gäbe.

Eine weitere Motivation ist von praktischer Natur: Die Semantik der puren Kernsprache ist gut untersucht und es existieren viele Techniken zum Korrektheitsnachweis von Programmtransformationen (siehe z. B. (Schmidt-Schauß et al., 2008c)). Diese Programmtransformationen werden in optimierenden Compilern verwendet um effizientiere Programme zu erzeugen. Wäre die Konservativität verletzt, so müsste die Korrektheit der Optimierungen erneut (mit der erweiterten Syntax bzw. der veränderten Semantik) geprüft werden und der bestehende Compiler wäre zunächst nicht als verlässlich (da er möglicherweise Semantik-verändernd übersetzt) einzustufen. Daher wäre es sehr wünschenswert, dass diese wohlbekannten Resultate und Techniken auch für die erweiterte (reale) Programmiersprache gelten.

Tatsächlich wurde in (Sabel & Schmidt-Schauß, 2012) die Konservativität der Erweiterung der puren funktionalen Kernsprache hin zu *CHF* gezeigt. Der Beweis ist dabei relativ aufwändig und erfolgt in mehreren Stufen. Der Kalkül *PF* besteht aus der rein funktionalen Teilsprache von *CHF*, d. h. die Syntax besteht nur aus den (weiterhin monomorph getypten) Ausdrücken:

$$\begin{aligned} e, e_i \in \mathit{Expr}_{PF} ::= {} & x \mid \lambda x.e \mid (e_1\ e_2) \mid c\ e_1 \ldots e_{\mathrm{ar}(c)} \mid \texttt{seq}\ e_1\ e_2 \\ & \mid \texttt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \texttt{in}\ e \\ & \mid \texttt{case}_T\ e\ \texttt{of}\ (c_{T,1}\ x_1 \ldots x_{\mathrm{ar}(c_{T,1})} \to e_1) \ldots\ (c_{T,|T|}\ x_1 \ldots x_{\mathrm{ar}(c_{T,|T|})} \to e_{|T|}) \end{aligned}$$

Die Auswertung wurde der Einfachheit halber direkt innerhalb des Kalküls *CHF* definiert, d. h. ein *PF*-Ausdruck e may-konvergiert genau dann, wenn der Prozess $y \overset{\text{main}}{\Longleftarrow} \texttt{seq}\ e\ (\texttt{return}\ ())$ in *CHF* may-konvergent ist. Da *PF*-Ausdrücke keine monadischen Operationen beinhalten, ist die Auswertung deterministisch und es genügt in *PF* die May-Konvergenz im Rahmen der kontextuellen Gleichheit zu betrachten (da Should-Konvergenz in diesem Fall mit der May-Konvergenz zusammenfällt). Die kontextuelle Äquivalenz $\sim_{PF}$ in *PF* verwendet nur jene Kontexte, die aus den Ausdrücken Expr_{PF} konstruierbar sind.

Das Ziel – der Konservativitätsbeweis – in (Sabel & Schmidt-Schauß, 2012) war es nun die folgende Implikation zu beweisen:

$$\text{Für alle } e_1, e_2 \in \mathit{Expr}_{PF} \text{ gilt: } e_1 \sim_{PF} e_2 \implies e_1 \sim_{CHF} e_2$$

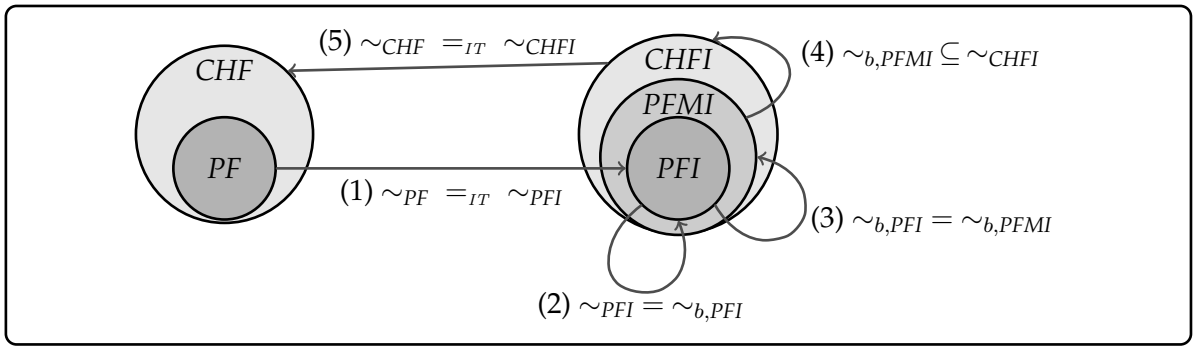


Abbildung 6.2: Beweisstruktur zur Konservativität von CHF gegenüber PF

Da ein direkter Beweis in den Kalkülen *CHF* und *PF* nicht gefunden wurde, wurde der Beweis in den Kalkülen mit unendlichen Bäumen erbracht. Abbildung 6.2 zeigt die Struktur des Beweises.

Der Kalkül *PFI* ist dabei der Kalkül *PF* mit unendlichen Bäumen jedoch ohne `letrec` (die Bindungen werden durch die Übersetzung IT entfaltet). Der Kalkül *PFMI* erweitert *PFI* um die monadischen Ausdrücke, die jedoch wie Konstruktoren behandelt werden. Beide Kalküle benutzen die call-by-name Auswertung auf unendlichen Bäumen.

Da die Übersetzungen $IT : CHF \to CHFI$ und $IT : PF \to PFI$ konvergenzäquivalent sind (dargestellt durch (1) und (5) in Abbildung 6.2), reicht es aus, die Implikation $e_1 \sim_{PFI} e_2 \implies e_1 \sim_{CHFI} e_2$ für alle unendlichen Bäume $e_1, e_2 \in PFI$ nachzuweisen (die Aussage $e_1 \sim_{PF} e_2 \implies e_1 \sim_{CHF} e_2$ folgt dann sofort). Um $e_1 \sim_{PFI} e_2 \implies e_1 \sim_{CHFI} e_2$ zu zeigen, wurde zunächst eine applikative Bisimulation $\sim_{b,PFI}$ im Kalkül *PFI* definiert und gezeigt, dass diese identisch zur kontextuellen Äquivalenz $\sim_{PFI}$ ist ((2) in Abbildung 6.2). Hierfür wurde Howes Methode (Howe, 1989, 1996) verwendet, die leicht adaptiert werden musste, da die Syntax coinduktiv definiert ist. Im nächsten Schritt wurde gezeigt, dass $e_1 \sim_{b,PFI} e_2$ äquivalent zu $e_1 \sim_{b,PFMI} e_2$ für alle *PFI*-Ausdrücke e_1, e_2 ist ((3) in Abbildung 6.2), wobei $\sim_{b,PFMI}$ die applikative Bisimulation in *PFMI* ist. Im letzten Schritt ((4) in Abbildung 6.2) wurde in einem rein syntaktischen Beweis gezeigt, dass aus $e_1 \sim_{b,PFMI} e_2$ auch $e_1 \sim_{CHFI} e_2$ für alle *PFMI*-Ausdrücke e_1, e_2 folgt.

Insgesamt lässt sich der Konservativitätsbeweis daher durch die Folgerungsschritte

(1) $e_1 \sim_{PF} e_2 \implies IT(e_1) \sim_{PFI} IT(e_2)$

(2) $IT(e_1) \sim_{PFI} IT(e_2) \implies IT(e_1) \sim_{b,PFI} IT(e_2)$

(3) $IT(e_1) \sim_{b,PFI} IT(e_2) \implies IT(e_1) \sim_{b,PFMI} IT(e_2)$

(4) $IT(e_1) \sim_{b,PFMI} IT(e_2) \implies IT(e_1) \sim_{CHFI} IT(e_2)$

(5) $IT(e_1) \sim_{CHFI} IT(e_2) \implies e_1 \sim_{CHF} e_2$

für alle $e_1, e_2 \in PF$ zusammenfassen, die zusammengesetzt $e_1 \sim_{PF} e_2 \implies e_1 \sim_{CHF} e_2$ ergeben.

6.4.1 Grenzen der Konservativität

In (Sabel & Schmidt-Schauß, 2012) wurde zudem untersucht, ob Concurrent Haskell erweitert um `unsafeInterleaveIO` ebenfalls konservativ bezüglich der reinen funktionalen Sprache ist. Hierbei wurde eine Grenze der Konservativität gefunden: Durch Hinzufügen von lazy Futures zu *CHF* erhält man einen Programmkalkül der keine konservative Erweiterung von *PF* ist. Lazy Futures $x \overset{lazy}{\Longleftarrow} e$ verhalten sich wie Futures, die jedoch noch nicht ausgewertet werden dürfen. Sobald (und ausschließlich in diesem Fall) ein anderer Thread jedoch den Wert einer lazy Future benötigt, wird die lazy Future $x \overset{lazy}{\Longleftarrow} e$ durch eine herkömmliche Future $x \Leftarrow e$ ersetzt.

Lazy Futures können mithilfe von `unsafeInterleaveIO` in Concurrent Haskell implementiert werden, indem der `future`-Aufruf mittels `unsafeInterleaveIO` verzögert wird:

```
lazyfuture :: IO a -> IO a
lazyfuture act = unsafeInterleaveIO (future act)
```

Sei *CHFL* die Erweiterung von *CHF* um lazy Futures, und $\sim_{CHFL}$ die entsprechende kontextuelle Gleichheit. Es lässt sich nachweisen, dass die Gleichheit $\texttt{seq}\ e_1\ e_2 \sim_{PF} \texttt{seq}\ e_2\ (\texttt{seq}\ e_1\ e_2)$ in *PF* gilt, da die Reihenfolge der Auswertung der Ausdrücke e_1 und e_2 nicht „beobachtbar" ist. Allerdings gilt diese Gleichung nicht mehr in *CHFL*: Für $e_1 = x$ und $e_2 = y$ unterscheidet der Kontext

$$\begin{aligned} C = z \overset{\mathsf{main}}{\Longleftarrow} \texttt{case}_{\mathrm{Bool}}\ [\cdot]\ \texttt{of}\ (\texttt{True} \to \bot)\ (\texttt{False} \to \texttt{return}\ \texttt{True}) \mid v\ \mathsf{m}\ \texttt{True} \\ \mid x \overset{lazy}{\Longleftarrow} \texttt{takeMVar}\ v\ \texttt{>>=}\ \lambda w.(\texttt{putMVar}\ v\ \texttt{False}\ \texttt{>>=}\ \lambda\_.\texttt{return}\ w) \\ \mid y \overset{lazy}{\Longleftarrow} \texttt{takeMVar}\ v\ \texttt{>>=}\ \lambda w.(\texttt{putMVar}\ v\ \texttt{False}\ \texttt{>>=}\ \lambda\_.\texttt{return}\ w) \end{aligned}$$

beide Ausdrücke, denn $C[\texttt{seq}\ x\ y]$ ist must-divergent, während $C[\texttt{seq}\ y\ x]$ should-konvergent ist. Informell ausgedrückt kann in *CHFL* die Reihenfolge der Auswertung beobachtet werden, da diese die Auswertung von lazy Futures anstoßen können.

Insgesamt lässt sich daher grob feststellen, dass (monomorph getyptes[1]) Concurrent Haskell eine konservative Erweiterung der funktionalen Kernsprache von Haskell ist, dass beliebiges Hinzufügen von `unsafeInterleaveIO` diese Konservativität verletzt, aber das Hinzufügen von Futures die Konservativität noch erhält.

6.5 Eine Abstrakte Maschine für CHF

Die Definition der Standardreduktion in CHF (siehe Abbildung 6.1c) ist relativ kompliziert. Die dabei verwendeten *LC*-Kontexte führen implizit eine Suche in der Menge der Bindungen durch und innerhalb der Ausdrücke muss auch entsprechend der leftmost-outermost Strategie gesucht werden. Kurzum ist es nicht völlig offensichtlich, wie ein Interpreter für *CHF* implementiert werden sollte. Daher wurde in (Sabel, 2012b) eine abstrakte Maschine als alternative operationale Semantik für *CHF* entworfen, die zum einen einfach zu implementieren ist

[1] Der Beweis für den polymorphen Fall steht noch aus.

und zum anderen als äquivalent zur ursprünglichen Semantik aus (Sabel & Schmidt-Schauß, 2011b) gezeigt wurde.

Die abstrakte Maschine wurde dabei in drei Schritten hergeleitet. Zunächst wurde die Maschine $M1$ definiert, die ausschließlich funktionale Ausdrücke (entsprechend dem *PF*-Kalkül) auswerten kann. Diese Maschine entspricht im Wesentlichen der Maschine in (Sestoft, 1997), wobei sie leicht angepasst wurde, damit auch `seq`- und `case`-Ausdrücke ausgewertet werden können. Der Zustand der Maschine $M1$ ist dabei ein Dreitupel $(\mathcal{H}, e, \mathcal{S})$, wobei $\mathcal{H}$ ein Heap ist, der Bindungen der Form $x \mapsto e$ enthält, e ist der aktuell auszuwertende Ausdruck und $\mathcal{S}$ ist ein Stack, auf dem sich der Evaluationskontext gemerkt wird. Ausdrücke haben in allen Maschinen wie üblich eine eingeschränktere Syntax als Ausdrücke in den Kalkülen *PF* und *CHF*: Argumente in Applikationen und Konstruktoranwendungen, das zweite Argument von `seq` und der Inhalt von MVars ist auf Variablen beschränkt. Im Stack werden sich dementsprechend nur Variablen oder `case`-Alternativen gemerkt. Genauer sind mögliche Einträge im Stack: $\#_{\mathsf{app}}(x)$ für das Argument x einer Anwendung $(e\ x)$, $\#_{\mathsf{seq}}(x)$ für das zweite Argument x eines `seq`-Ausdrucks `seq` $e\ x$, $\#_{\mathsf{case}}(alts)$ für die Alternativen eines `case`-Ausdrucks und $\#_{\mathsf{heap}}(x)$, falls der Ausdruck e einer Heapbindung $x \mapsto e$ aktuell auf der Maschine ausgewertet wird.

Die Transitionsrelation $\xrightarrow{M1}$ überführt Maschinenzustände in Maschinenzustände. Sie verwendet im Wesentlichen Regeln zum Finden des nächstens Redex, z. B. eine Regel $(\mathcal{H}, (e\ x), \mathcal{S}) \xrightarrow{M1} (\mathcal{H}, e, x : \mathcal{S})$, und Regeln die β-, case, und seq-Reduktion durchführen, z. B. $(\mathcal{H}, \lambda x.e, \#_{\mathsf{app}}(y) : \mathcal{S}) \to (\mathcal{H}, e[y/x], \mathcal{S})$.

Die nächste Stufe ist die Maschine $IOM1$, welche die Maschine $M1$ um Speicherplätze (in Form von MVars) und monadische Operationen zum Erstellen und zum Zugriff auf die Speicherplätze erweitert. Der Zustand der Maschine $IOM1$ ist ein Fünf-Tupel $(\mathcal{H}, \mathcal{M}, e, \mathcal{S}, \mathcal{I})$, wobei $\mathcal{H}$, e und $\mathcal{S}$ die bekannten Komponenten der Maschine $M1$ sind, $\mathcal{M}$ eine Menge von MVars ist und $\mathcal{I}$ ein sogenannter IO-Stack ist, der sich im Wesentlichen den Kontext für die monadischen Berechnungen merkt (er entspricht daher den *MC*-Kontexten im *CHF*-Kalkül aus Abbildung 6.1b). Die Transitionsrelation $\xrightarrow{IOM1}$ der Maschine $IOM1$ verwendet sämtliche Transitionen der Maschine $M1$ wieder. Diese werden angewendet, solange der Stack $\mathcal{S}$ nicht leer ist und der Ausdruck kein funktionaler Wert ist. Neue Regeln werden hinzugefügt für das Erzeugen und den Zugriff auf MVars, sowie für die Redex-Suche innerhalb der monadischen Operatoren. Z. B. ist $(\mathcal{H}, \mathcal{M}, x >\!\!>\!\!= y, [], \mathcal{I}) \xrightarrow{IOM1} (\mathcal{H}, \mathcal{M}, x, [], \#_{>\!\!>\!\!=}(y) : \mathcal{I})$ die Regel, die dafür sorgt, dass in einer monadischen Sequenz $x >\!\!>\!\!= y$ zunächst x ausgeführt wird bevor y ausgeführt werden kann. Dementsprechend gibt es eine Regel $(\mathcal{H}, \mathcal{M}, \texttt{return}\ x, \#_{>\!\!>\!\!=}(y) : \mathcal{I}) \xrightarrow{IOM1} (\mathcal{H}, \mathcal{M}, (y\ x), \mathcal{I})$, die durch Anwendung des ersten monadischen Gesetzes den >>= -Operator auswertet.

Im dritten Schritt wird die Maschine $IOM1$ um Nebenläufigkeit erweitert. Die so entstandene Maschine $CIOM1$ verwendet als Zustand das Dreitupel $(\mathcal{H}, \mathcal{M}, \mathcal{T})$, wobei $\mathcal{H}$ der Heap, $\mathcal{M}$ die Menge der MVars ist und $\mathcal{T}$ eine Menge von Threads (genauer Futures) ist. Jeder Thread ist dabei ein Viertupel $(x, e, \mathcal{S}, \mathcal{I})$ welches aus dem Namen x der Future, dem auszuwertenden Ausdruck e, einem Stack $\mathcal{S}$ und einem IO-Stack $\mathcal{I}$ besteht. Analog zu *CHF* gibt es einen

ausgezeichneten Main-Thread. Die Übergangsrelation $\xrightarrow{CIOM1}$ verwendet in großen Teilen die Relation $\xrightarrow{IOM1}$ der Maschine $IOM1$ wieder: Ein Thread wird selektiert und dieser darf anschließend einen Schritt bezüglich der Maschine $IOM1$ durchführen. Zwei wirklich neue Regeln beinhaltet die Maschine $CIOM1$: Eine Regel zur Auswertung der `future`-Operation, die einen neuen Thread in die Menge $\mathcal{T}$ einfügt, und eine Regel zum Entfernen eines erfolgreich ausgewerteten Threads: In diesem Fall wird das Ergebnis als Heapbindung im Heap $\mathcal{H}$ gesichert.

Basierend auf der Maschinentransition wurde in (Sabel, 2012b) die May- und Should-Konvergenz von Maschinenzuständen definiert und eine kontextuelle Äquivalenz für Ausdrücke definiert, indem das Konvergenzverhalten der Maschine mit dem Startzustand $(\emptyset, \emptyset, \{(x, C[e], [], [])^{\mathsf{main}}\})$ für alle Kontexte C beobachtet wird.

In (Sabel, 2012b) wurde nachgewiesen, dass die Konvergenzbegriffe basierend auf der Maschine und die Konvergenzbegriffe aus dem *CHF*-Kalkül übereinstimmen, d. h. es gilt Konvergenzäquivalenz. Dies zeigt daher, dass die Maschine CIOM1 ein korrekter Evaluator für den *CHF*-Kalkül ist (vgl. Abschnitt 3.1).

7

Eine zweiwertige Logik für eine funktionale Call-by-value Sprache mit kontextueller Äquivalenz

In diesem Kapitel fassen wir die Untersuchung aus (Sabel & Schmidt-Schauß, 2013) zu einer Programmlogik für eine strikte funktionale Programmiersprache zusammen. Auch in diesem Rahmen wird die kontextuelle Gleichheit eine wesentliche Rolle spielen.

Nach einem Überblick über Programmlogiken und der Motivation und Klassifizierung unseres Ansatzes in Abschnitt 7.1 wird in Abschnitt 7.2 zunächst die Syntax und Semantik der funktionalen Programmiersprache erläutert. In Abschnitt 7.3 wird die kontextuelle Semantik der Sprache definiert und Resultate zur Konservatität der kontextuellen Gleichheit gegenüber Programmerweiterungen werden erörtert. In Abschnitt 7.4 wird die Syntax und Semantik der logischen Formeln erläutert, um schließlich in Abschnitt 7.5 die in (Sabel & Schmidt-Schauß, 2013) herausgearbeiteten Eigenschaften der Logik zu erörtern.

7.1 Motivation

Programmlogiken dienen dazu, Eigenschaften von Programmen formal zu erfassen und Aussagen über Programme formal herzuleiten. Mithilfe von (halb-) automatischen Theorembeweisern können solche Aussagen (halb-) automatisch bewiesen werden. Prominente Beispiele sind die Theorembeweiser Coq[1] (Bertot & Castéran, 2004), der auf konstruktiver Typentheorie (Coquand & Huet, 1988) basiert, Isabelle/HOL[2] (Nipkow et al., 2002), der eine Prädikatenlogik höherer Ordnung verwendet, oder auch ACL2[3], der auf einer Prädikatenlogik erster Ordnung

[1] *http://coq.inria.fr/*
[2] *http://www.cl.cam.ac.uk/research/hvg/isabelle/*
[3] *http://www.cs.utexas.edu/∼moore/acl2*

basiert. Ein weiteres solches System ist VeriFun[4] (Walther & Schweitzer, 2005; Schlosser et al., 2007; Walther, 1994), welches auf einer Prädikatenlogik erster Ordnung über einer strikten funktionalen Programmiersprache basiert.

Viele solcher Ansätze verwenden als Gleichheitssemantik von Programmen die Konvertibilität von Ausdrücken (vgl. Abschnitt 2.2.1), bzw. den Vergleich der Resultate der Auswertung. Oft verwendet die Logik dabei die Annahme, dass alle Programme terminieren, d. h. bevor die Terminierung der Programme nicht gezeigt ist, kann keine andere Eigenschaft über die Programme in der Logik bewiesen werden. Allerdings sind nichtterminierende Programme aus zweierlei Gründen sehr üblich, d. h. sie treten in der realen Welt auf: Zum einen gibt es Programme, deren mögliche Nichtterminierung in Kauf genommen werden muss, wie z. B. Interpreter von Turing-mächtigen Programmiersprachen. Auch über solche Interpreter möchte man Aussagen in der Logik (bzw. im Theorembeweiser) formulieren und beweisen können. Ein anderer Fall für nichtterminierende Programme sind partiell definierte Funktionen, wie z. B. die `tail`-Funktion auf Listen, die keine Spezifikation für die leere Liste als Argument besitzt. Semantisch werden die undefinierten Fälle mit Nichtterminierung gleichgesetzt. Zumindest für den Fall der partiell definierten Funktionen bieten Theorembeweiser (und die entsprechenden Logiken) Auswege, um sie trotz der Terminierungsanforderung zu behandeln. Übersichten hierzu sind beispielsweise in (Cheng & Jones, 1991; Schieder & Broy, 1999; Farmer, 2004; Bove et al., 2011) zu finden). Die wesentlichen Methoden lassen sich dabei wie folgt klassifizieren:

1. Nur totale und terminierende Funktionen werden von der Logik erfasst. Partielle Funktionen werden wie totale Funktionen behandelt, indem sie durch explizite Angabe des Definitionsbereichs eingeschränkt werden.

2. Partialität wird in die Logik mit aufgenommen, indem eine dreiwertige Logik (wahr, falsch und undefiniert) verwendet wird.

3. Partielle Funktionen werden durch lose Spezifikation in die totale Logik integriert: Wenn eine Funktion f auf ein nicht-definiertes Argument a angewendet wird, so wird $f(a)$ als Ausdruck angesehen, der jeden Wert (des Wertebereichs von f) annehmen kann.

4. Nichtterminierung wird als eigener Wert $\bot$ in die Programmiersprache, aber nicht in die Logik integriert.

Alle Ansätze führen jedoch zu unterschiedlichen Logiken und entsprechend unterschiedlichen Theoremen der Logik. Eine ausführliche Diskussion zu den verschiedenen Varianten und deren semantischen Konsequenzen ist in (Sabel & Schmidt-Schauß, 2013) zu finden. Wir gehen an dieser Stelle nur auf wesentliche Eigenschaften und Unterschiede ein. Das explizite Definieren des Definitionsbereichs (der erste Ansatz) ist aufwändig, da die Angabe des genauen Definitionsbereichs oft nicht trivial ist und Beweise über partielle Funktionen stets auch Aussagen über den Definitionsbereich erfordern. Echt nichtterminierende Funktionen werden von

[4] *http://www.inferenzsysteme.informatik.tu-darmstadt.de/verifun/*

diesem Ansatz im Allgemeinen nicht erfasst. Z. B. kann eine Aussage der Art „Für die Eingabe XYZ terminiert die Funktion f nicht“ nicht in einer solchen Logik formuliert werden.

Der zweite Ansatz, d. h. die Verwendung einer dreiwertigen Logik, lässt einerseits Freiheiten innerhalb der logischen Semantik, da „undefiniert“ als logischer Wert behandelt werden muss und je nach Festlegung der Semantik der aussagenlogischen Operatoren andere Theoreme gelten. Zudem gelten aufgrund der Dreiwertigkeit die Schlussregeln der klassischen Logik nicht.

Das Verwenden von loser Spezifikation (dritter Ansatz) kann problematische Effekte aufweisen. Z. B. können für einelementige Definitionsbereiche partielle Funktionen nicht von totalen Funktionen unterschieden werden. Ein anderer Effekt ist, dass die Logik Inkonsistenzen aufweisen kann, wenn lose Spezifikation auch für echte Nichtterminierung verwendet wird, wie z. B. das folgende leicht angepasste Beispiel aus (Giesl, 2001) zeigt: Sei f auf (strikten) Peano-Zahlen[5] definiert als $f\ x := S\ (f\ x)$. Dann terminiert $(f\ z)$ für keine Eingabe z und f ist daher eine nichtterminierende Funktion. Gleichermaßen erfordert die lose Spezifikation jedoch auch, dass $f\ 0$ als „irgendeine“ Peanozahl a interpretiert wird. Aus der Definition von f lässt sich dann folgern $a = f\ 0 = S\ (f\ 0) = S\ a$ was zur inkonsistenten Folgerung $a = S\ a$ (alle Peanozahlen sind gleich) führt. Wenn lose Spezifikationen nur für undefinierte Pattern verwendet werden (wie z. B. bei der $tail$-Funktion), treten diese Inkonsistenzen nicht auf. Allerdings werden manche Aussagen als Theoreme behandelt, deren Gültigkeit eher nicht intuitiv ist. Ein solches Beispiel ist die Formel

$$tail(tail\ \texttt{Nil}) = tail\ \texttt{Nil} \implies tail\ \texttt{Nil} = \texttt{Nil}$$

Da sowohl $tail(tail\ \texttt{Nil})$ als auch $(tail\ \texttt{Nil})$ außerhalb des Definitionsbereichs von $tail$ liegen, wird für beide Ausdrücke irgendein Wert angenommen, der nicht notwendigerweise gleich ist. Daher wird die linke Seite der Implikation als falsch interpretiert, wodurch die Formel insgesamt als Theorem bewiesen werden kann.

Der vierte Ansatz erlaubt Freiheiten, wie die Konstante $\bot$ innerhalb von Gleichungen behandelt wird. Logiken mit einer sogenannten *schwachen Gleichheit* interpretieren jede Gleichung $\bot = e$ bzw. $e = \bot$ als falsch (siehe z. B. (Farmer, 1993)), wohin gegen *starke Gleichheit* erfordert, dass Gleichungen $\bot = \bot$ als wahr interpretiert werden.

Ziel der Untersuchung in (Sabel & Schmidt-Schauß, 2013) war es, die Gleichungen in einer solchen Programmlogik aus Sicht der Programmsemantik zu sehen, d. h. kontextuelle Äquivalenz als Semantik der Gleichungen innerhalb der Logik zu verwenden.

Für die kontextuelle Gleichheit ist die Behandlung von nichtterminierenden Ausdrücken unproblematisch, da diese genau wie jeder andere Ausdruck behandelt werden. Partiell definierte Funktionen können ebenfalls problemlos (ohne spezielle Erweiterung der Semantik) behandelt werden, indem die entsprechenden `case`-Alternativen für undefinierte Werte einen

[5] Peano-Zahlen werden dabei aus der Konstante 0 und dem einstelligen Konstruktor S konstruiert

Typen: $\tau, \tau_i \in \mathcal{T} ::= X \mid (\tau_1 \to \tau_2) \mid (K\ \tau_1 \dots \tau_{\mathrm{ar}(K)})$, wobei $K \in \mathcal{K}, X \in TVar$

Typschemata: $\forall \mathcal{X}.\tau$, wobei $\tau \in \mathit{Typ}$ und $\mathcal{X}$ ist die Menge aller Typvariablen in τ

Ausdrücke:
$$e, e_i \in \mathcal{E} ::= x \mid f \mid (e_1\ e_2) \mid \lambda x.e \mid (c_i\ e_1 \dots e_{\mathrm{ar}(c_i)})$$
$$\mid (\mathtt{case}_K\ e\ ((c_1\ x_{1,1} \dots x_{1,\mathrm{ar}(c_1)}) \to e_1)$$
$$\dots$$
$$((c_n\ x_{n,1} \dots x_{n,\mathrm{ar}(c_n)}) \to e_n)) \text{ mit } \{c_1, \dots, c_n\} = D_K$$
wobei $f \in \mathcal{F}$, $K \in \mathcal{K}, c_i \in \mathcal{D}, D_K \subseteq \mathcal{D}$ und $x, x_i \in \mathit{Var}$

Werte: $v, v_i \in \mathit{Val} ::= x \mid \lambda x.e \mid (c\ v_1 \dots v_n)$ wobei $e \in \mathit{Expr}$, $x \in \mathit{Var}$

Kontexte:
$$C \in \mathcal{C} ::= [\cdot] \mid \lambda x.C \mid (C\ e) \mid (e\ C) \mid (c_i\ e_1 \dots e_j\ C\ e_{j+2} \dots e_{\mathrm{ar}(c_i)})$$
$$\mid\ (\mathtt{case}_K\ C\ alts) \mid (\mathtt{case}_K e\ \dots (c_i\ x_1 \dots x_{\mathrm{ar}(c_i)} \to C) \dots)$$
wobei $e, e_i \in \mathit{Expr}$, $x, x_i \in \mathit{Var}$

Reduktionskontexte:
$$R \in \mathcal{R} ::= [\cdot] \mid (R\ e) \mid (v\ R) \mid \mathtt{case}_K\ R\ alts \mid (c\ v_1\ \dots\ v_{i-1}\ R\ e_{i+1}\ \dots\ e_n)$$
wobei $e, e_i \in \mathit{Expr}$, $v, v_i \in \mathit{Val}$

Abbildung 7.1: Syntax von Typen, Ausdrücken, Werten, Kontexten und Reduktionskontexten über einer Signatur $(\mathcal{F}, \mathcal{K}, \mathcal{D})$, einer Menge von Variablen Var, und einer Menge von Typvariablen $TVar$.

nichtterminierenden Ausdruck (welcher im Folgenden durch $\bot$ repräsentiert wird) zurückliefern. Z. B. kann `tail` durch den Ausdruck

$$\lambda xs.\mathtt{case}_{List}\ xs\ (\mathtt{Nil} \to \bot)\ ((y : ys) \to ys)$$

definiert werden. Im Groben entspricht dieser Ansatz der vierten Alternative, wobei der Wert $\bot$ nicht hinzugefügt werden muss, er ist bereits als nichtterminierendes Programm in der Sprache der Programme vorhanden. Die Gleichheit ist dabei starke Gleichheit, da $\bot = \bot$ als wahr interpretiert wird. Ebenso wird die Formel

$$\mathit{tail}(\mathit{tail}\ \mathtt{Nil}) = \mathit{tail}\ (\mathtt{Nil}) \implies \mathit{tail}\ \mathtt{Nil} = \mathtt{Nil}$$

bezüglich einer solchen Logik als falsch interpretiert, da $\mathit{tail}(\mathit{tail}\ \mathtt{Nil})$ und $\mathit{tail}\ \mathtt{Nil}$ jeweils äquivalent zu $\bot$ sind, und daher die linke Seite der Implikation als wahr interpretiert wird.

Die in (Sabel & Schmidt-Schauß, 2013) eingeführte Logik LMF orientiert sich an der in VeriFun verwendeten Prädikatenlogik erster Ordnung, jedoch erweitert um Funktionen höherer Ordnung, und ohne die Anforderung der Terminierung aller Ausdrücke in Formeln. Die Logik LMF ist zweistufig aufgebaut. Auf oberer Ebene befinden sich die logischen Formeln einer Prädikatenlogik, die All- und Existenzquantifizierung über Werte der Programmiersprache erlaubt. Innerhalb der Formeln (auf unterer Ebene) können Programme (Ausdrücke) einer strikten funktionalen Programmiersprache verwendet werden.

7.2 Die funktionale Programmiersprache

Wir beschreiben zunächst die funktionale Programmiersprache. Die Sprache verwendet Ausdrücke und definierte Funktionen. Ausdrücke sind über einer Signatur $(\mathcal{F}, \mathcal{K}, \mathcal{D})$ aufgebaut, wobei $\mathcal{F}$ eine Menge von Funktionssymbolen, $\mathcal{K}$ eine Menge von Typkonstruktoren und $\mathcal{D}$ eine Menge von Datenkonstruktoren ist. Dabei gehört jeder Datenkonstruktor $c_i \in \mathcal{D}$ zu genau einem Typkonstruktor $K \in \mathcal{K}$. Typ- und Datenkonstruktoren haben eine feste Stelligkeit und zu jedem Typkonstruktor gibt es mindestens einen Datenkonstruktor. Die Syntax von Typen, Ausdrücken und Werten über einer solchen Signatur ist in Abbildung 7.1 dargestellt. Ausdrücke sind dabei monomorph (ohne Typschemata) mit einem üblichen monomorphen Typsystem zu typisieren. Da die Definition der Funktionssymbole noch fehlt, werden *Programme* definiert:

Definition 7.2.1. *Ein* Programm $\mathcal{P}$ *besteht aus einer Signatur* $(\mathcal{F}, \mathcal{K}, \mathcal{D})$ *(wobei* $\mathcal{K} \neq \emptyset$*) und der Menge von Paaren* $Def_{\mathcal{F}} := \{(f, d_f) \mid f \in \mathcal{F}\}$*, wobei* d_f *ein geschlossener Wert ist und* Definitionsausdruck von f *genannt wird. In* d_f *dürfen nur die Funktionssymbole aus* $\mathcal{F}$ *vorkommen. Anstelle von* (f, d_f) *schreiben wir auch* $f = d_f$*. Die Definitionsausdrücke sind polymorph mit Typschemata (Abbildung 7.1) getypt.*

Für ein Programm $\mathcal{P}$ werden Ausdrücke, Kontexte und Typen über einem solchen Programm als $\mathcal{P}$-Ausdrücke, $\mathcal{P}$-Kontexte und $\mathcal{P}$-Typen bezeichnet.

Für die Logik LMF und deren funktionale Kernsprache wurde daher nicht eine allgemeine Menge von Datenkonstruktoren (und Funktionssymbolen) angenommen, sondern diese durch die Signatur fixiert. Der Grund dafür liegt darin, dass man (automatische) Beweise für logische Formeln im Allgemeinen für eine feste Sprache führen will. Trotzdem erwartet man, dass die Aussagen durch Erweiterung der Programme nicht verändert werden (d. h. die Logik sollte konservativ bezüglich Programmerweiterungen sein).

Definition 7.2.2. *Ein Programm* $\mathcal{P}' = ((\mathcal{F}', \mathcal{K}', \mathcal{D}'), Def_{\mathcal{F}'})$ erweitert *das Programm* $\mathcal{P} = ((\mathcal{F}, \mathcal{K}, \mathcal{D}), Def_{\mathcal{F}})$ *(geschrieben als* $\mathcal{P}' \sqsupseteq \mathcal{P}$*), wenn* $\mathcal{F}' \supseteq \mathcal{F}, \mathcal{K}' \supseteq \mathcal{K}, \mathcal{D}' \supseteq \mathcal{D}, Def'_{\mathcal{F}'} \supseteq Def_{\mathcal{F}}$*, sodass* $\mathcal{D}' = \mathcal{D} \cup \bigcup_{K \in \mathcal{K}' \setminus \mathcal{K}} D_K$*.*

Die operationale Semantik (die Standardreduktion $\xrightarrow{LMF}$) implementiert die call-by-value-Reduktion und ist in Abbildung 7.2 definiert. Die Standardreduktion ist eindeutig, daher handelt es sich um einen deterministischen Programmkalkül. Ein geschlossener Ausdruck e ist may-konvergent ($e{\downarrow}$), wenn er mit Standardreduktionen in einen Wert überführt wird (d. h. $e \xrightarrow{LMF,*} v$, wobei v ein Wert ist).

7.3 Kontextuelle Äquivalenz von Ausdrücken

Für Ausdrücke werden zwei Formen von kontextueller Äquivalenz für Ausdrücke verwendet. Die erste Gleichheit betrachtet nur das Programm, welches die Ausdrücke verwenden (daher ist sie eine lokale Gleichheit), während die zweite Variante alle Erweiterungen des Programms miteinbezieht und daher eine globale Gleichheit ist. Im Unterschied zu den kontex-

Reduktionsregeln:

(beta) $((\lambda x.e)\ v) \to e[v/x]$ where $v \in \mathit{Val}$

(delta) $f :: T \to d_f$ wenn $f = d_f :: T' \in \mathit{Def}_{\mathcal{F}}$
wobei mit der Reduktion eine Typinstantiierung $\rho(d_f)$ durchgeführt wird, sodass $\rho(T') = T$

(case) $(\texttt{case}\ (c\ v_1 \ldots v_n)\ \ldots((c\ y_1 \ldots y_n) \to e)\ldots) \to e[v_1/y_1, \ldots, v_n/y_n]$
where $v_1, \ldots, v_n \in \mathit{Val}$

Standardreduktion: $R[e_1] \xrightarrow{sr} R[e_2]$ falls $e_1 \to e_2$ mit einer Reduktionsregel und $R \in \mathcal{R}$.

Abbildung 7.2: Reduktionsregeln und Standardreduktion für $\mathcal{P}$-Ausdrücke ($\mathcal{P} = ((\mathcal{F}, \mathcal{K}, \mathcal{D}), \mathit{Def}_{\mathcal{F}})$)

tuellen Gleichheiten in call-by-need Lambda-Kalkülen werden zudem – wie für call-by-value Kalküle üblich – nur schließende Kontexte betrachtet.

Definition 7.3.1. *Die lokale kontextuelle Präordnung $\leq_{\mathcal{P}}$ für $\mathcal{P}$-Ausdrücke ist definiert durch: Für $\mathcal{P}$-Ausdrücke e_1, e_2 vom gleichen Typ T gilt $e_1 \leq_{\mathcal{P}} e_2$ genau dann, wenn für alle $\mathcal{P}$-Kontexte $C \in \mathcal{C}_{T,T'}$, sodass $C[e_1]$ und $C[e_2]$ geschlossene Ausdrücke sind, gilt $C[e_1]{\downarrow} \implies C[e_2]{\downarrow}$.*

Die (globale) kontextuelle Präordnung $\leq_{\forall\mathcal{P}}$ für $\mathcal{P}$-Ausdrücke ist definiert durch: Für $\mathcal{P}$-Ausdrücke e_1, e_2 vom gleichen Typ T gilt $e_1 \leq_{\forall\mathcal{P}} e_2$ genau dann, wenn für alle Erweiterungen $\mathcal{P}' \sqsupseteq \mathcal{P}$ und alle $\mathcal{P}'$-Kontexte $C \in \mathcal{C}_{T,T'}$, sodass $C[e_1]$ und $C[e_2]$ geschlossene Ausdrücke sind, gilt $C[e_1]{\downarrow} \implies C[e_2]{\downarrow}$.

Die kontextuellen Gleichheiten $\sim_{\mathcal{P}}$ und $\sim_{\forall\mathcal{P}}$ sind definiert durch: $\sim_{\mathcal{P}} := \leq_{\mathcal{P}} \cap \geq_{\mathcal{P}}$ und $\sim_{\forall\mathcal{P}} := \leq_{\forall\mathcal{P}} \cap \geq_{\forall\mathcal{P}}$.

In (Sabel & Schmidt-Schauß, 2013) wurde als wichtiges Resultat gezeigt, dass die lokale und die globale kontextuelle Äquivalenz zusammenfallen, oder anders ausgedrückt, dass die lokale kontextuelle Äquivalenz konservativ bezüglich Programmerweiterungen ist.

Theorem 7.3.2 ((Sabel & Schmidt-Schauß, 2013))**.** *$\leq_{\mathcal{P}} = \leq_{\forall\mathcal{P}}$ und $\sim_{\mathcal{P}} = \sim_{\forall\mathcal{P}}$*

Der Beweis dieser Aussage in (Sabel & Schmidt-Schauß, 2013) erfolgt schrittweise. Wir erläutern die einzelnen Schritte grob.

Zunächst wird ein sogenanntes CIU-Lemma für die lokale kontextuelle Äquivalenz nachgewiesen, welches besagt, dass es ausreicht, die Konvergenzerhaltung bzgl. aller Reduktionskontexte und schließenden Wertsubstitutionen zu betrachten, um kontextuelle Äquivalenz zu schließen. Für den entsprechenden Beweis wurden zum einen ähnliche Methoden wie im Beweis des allgemeinen Kontextlemmas (vgl. Abschnitt 4.1) als auch die Methode der beobachtungskorrekten Übersetzungen (vgl. Abschnitt 3.1) verwendet.

Im nächsten Schritt kann dieses CIU-Lemma noch verfeinert werden: Die betrachteten Reduktionskontexte und Wertsubstitutionen können so eingeschränkt werden, dass sie keine Funktionssymbole enthalten. Die Begründung hierfür ist, dass in gegebenen konvergierenden Reduktionsfolgen die Funktionssymbole direkt durch deren Definitionsausdrücke (iterativ entfaltet bis zu einer festen Tiefe) ersetzt werden können. D. h. gibt es einen Reduktionskontext und eine Wertsubstitution, welche Funktionssymbole enthalten und die kontextuelle

Gleichheit zweier Ausdrücke widerlegen, dann können durch die beschriebene Konstruktion auch ein Reduktionskontext bzw. eine Wertsubstitution erzeugt werden, die keine Funktionssymbole enthalten und ebenfalls die kontextuelle Gleichheit der Ausdrücke widerlegen. Daraus folgt schon die Konservativität bzgl. Programmerweiterungen, die neue Funktionssymbole hinzufügen, aber die Mengen der Typ- und Datenkonstruktoren unverändert lassen.

Schließlich wird das CIU-Lemma verwendet, um zu zeigen, dass neue Typ- und Datenkonstruktoren ebenfalls die Gleichheit nicht verändern können: Die gegebene Situation dabei ist die Reduktionssequenz beginnend mit einem Ausdruck der Form $R[\sigma(e)]$ und endend mit einem Wert v. Dabei ist e ein $\mathcal{P}$-Ausdruck, R ist ein $\mathcal{P}'$-Reduktionskontext, und σ ist eine schließende Wertsubstitutionen mit $\mathcal{P}'$-Werten, wobei $\mathcal{P}'$ das Programm $\mathcal{P}$ erweitert und R und σ keine Funktionssymbole enthalten. Informell gesagt, können Auswirkungen von R und σ auf den (möglicherweise offenen) Ausdruck e nur an irrelevanten Positionen auftreten, da e einen $\mathcal{P}$-Typ hat. In (Sabel & Schmidt-Schauß, 2013) wird eine Approximationsreduktion verwendet, die genau solche Unterausdrücke von $\mathcal{P}'$-Typen weg transformiert, indem sie die Unterausdrücke lokal auswertet und bei Nichtterminierung bzw. bei ausreichender Tiefe durch $\bot$-Konstanten ersetzt

Als weitere Resultate bezüglich kontextueller Gleichheit, wurde eine Reihe von Programmtransformationen als korrekt gezeigt, insbesondere die drei Reduktionsregeln (case), (beta) und (delta).

7.4 Die Logik LMF

Auf der Ebene der Logik LMF sind $\mathcal{P}$-Formeln durch die folgende Grammatik definiert, wobei e_i $\mathcal{P}$-Ausdrücke sind und T monomorphe Typen sind.

$$F, F_i \in F_P ::= \mathsf{tt} \mid \mathsf{ff} \mid (e_1 = e_2) \mid F_1 \vee F_2 \mid F_1 \wedge F_2 \mid \neg F \mid \forall x :: T.F \mid \exists x :: T.F$$

Als Abkürzungen verwenden wir auch $F_1 \implies F_2$ anstelle von $\neg F_1 \vee F_2$. Atome der Logik sind daher die Wahrheitswerte tt und ff sowie Gleichungen $(e_1 = e_2)$. Die Quantoren $\forall$ und $\exists$ quantifizieren über monomorph getypte Variablen der Programmiersprache. Da in call-by-value Sprachen Variablen nur durch Werte instantiiert werden, qunatifizieren die Quantoren dementsprechend nur über Werte. Wir definieren die Semantik einer geschlossenen $\mathcal{P}$-Formel, wobei erneut zwei Begriffe verwendet werden: Zum einen die lokale Gültigkeit der Formel innerhalb des Programms $\mathcal{P}$ und zum anderen die Gültigkeit für alle Programmerweiterungen von $\mathcal{P}$ (was als $\mathcal{P}$-Tautologie bezeichnet wird):

Definition 7.4.1. *Sei $\mathcal{M}_{\mathcal{P},T}$ die Menge aller geschlossenen $\mathcal{P}$-Werte vom (monomorphen) Typ T.*

Eine geschlossene (monomorphe) $\mathcal{P}$-Formel F ist $\mathcal{P}$-gültig *(notiert als $LMF(\mathcal{P}) \models F$), wenn $I(F) = \mathsf{tt}$ gilt, wobei*

$$
\begin{array}{lll}
I(\mathsf{tt}) & = \mathsf{tt} & I(\mathsf{ff}) \quad = \mathsf{ff} \\
I(A \wedge B) & = I(A) \wedge I(B) & I(A \vee B) = I(A) \vee I(B) \qquad I(\neg A) = \neg I(A) \\
I(e_1 = e_2) & = \mathsf{tt}, \textit{wenn } e_1 \sim_{\mathcal{P}} e_2 & \\
I(e_1 = e_2) & = \mathsf{ff}, \textit{wenn } e_1 \not\sim_{\mathcal{P}} e_2 & \\
I(\forall x :: T.F) = & \begin{cases} \mathsf{tt}, \textit{wenn für alle } a \in \mathcal{M}_{\mathcal{P},T} : I(F[a/x]) = \mathsf{tt} \\ \mathsf{ff}, \textit{sonst} \end{cases} & \\
I(\exists x :: T.F) = & \begin{cases} \mathsf{tt}, \textit{wenn es ein } a \in \mathcal{M}_{\mathcal{P},T} \textit{ gibt, mit} : I(F[a/x]) = \mathsf{tt} \\ \mathsf{ff}, \textit{sonst} \end{cases} &
\end{array}
$$

Eine geschlossene (monomorphe) $\mathcal{P}$-Formel F ist eine $\mathcal{P}$-Tautologie *(notiert als $LMF(\forall P) \models F$) genau dann, wenn für alle Erweiterungen $\mathcal{P}' \sqsupseteq \mathcal{P}$ die Formel F $\mathcal{P}'$-gültig ist.*

Da die $\mathcal{P}$-Gültigkeit einer Formel mithilfe der Funktion I bestimmt wird, sind die lokalen Logiken $LMF(\mathcal{P})$ vollständig, d. h. für jede $\mathcal{P}$-Formel F gilt $LMF(\mathcal{P}) \models F$ oder $LMF(\mathcal{F}) \models \neg F$. Dies gilt jedoch nicht für den Tautologiebegriff, d. h. die globalen Logiken $LMF(\forall\mathcal{P})$, da durchaus Formeln F und Programme $\mathcal{P}' \sqsupseteq \mathcal{P}$ existieren, sodass F $\mathcal{P}$-gültig ist und $\neg F$ $\mathcal{P}'$-gültig ist, d. h. weder $LMF(\forall\mathcal{P}) \models F$ noch $LMF(\forall\mathcal{P}) \models \neg F$ gilt.

7.5 Eigenschaften der Logik LMF

Der wesentliche Begriff für Formeln ist die $\mathcal{P}$-Tautologie, deren Definition impliziert, dass sich die Logik LMF monton bezüglich Programmerweiterungen verhält: Jede $\mathcal{P}$-Tautologie ist gültig in allen Programmerweiterungen $\mathcal{P}' \sqsupseteq \mathcal{P}$ und insbesondere daher auch eine $\mathcal{P}'$-Tautologie.

Eine wichtige Fragestellung ist jedoch auch, ob $\mathcal{P}$-Gültigkeit bereits die $\mathcal{P}$-Tautologieeigenschaft impliziert. Diese Eigenschaft für eine Klasse von Formeln nennen wir *Konservativität bezüglich Programmerweiterungen*:

Definition 7.5.1. *Sei $\mathcal{P}$ ein Programm. Eine Klasse X von $\mathcal{P}$-Formeln ist genau dann* konservativ bezüglich Programmerweiterungen, *wenn für jede Formel $F \in X$ aus $LMF(\mathcal{P}) \models F$ auch $LMF(\forall\mathcal{P}) \models F$ folgt.*

Die Klasse aller Formeln ist nicht konservativ bezüglich Programmerweiterungen (siehe (Sabel & Schmidt-Schauß, 2013)). Allerdings konnte die Konservativität für verschiedene Klassen von Formeln in (Sabel & Schmidt-Schauß, 2013) nachgewiesen werden. Dabei ist Theorem 7.3.2 als wichtigste Vorarbeit zu nennen. Z. B. folgt aus der Semantik der Formeln und Theorem 7.3.2 sofort, dass die Menge der Quantor-freien geschlossenen Formeln konservativ bezüglich Programmerweiterungen ist.

Im Folgenden erläutern wir weitere Klassen von Formeln.

Die Klasse der *AEQ-Formeln* (allquantifzierte Gleichungen) umfasst alle geschlossenen Formeln der Form $\forall x_1 :: T_1, \ldots, x_n :: T_n.e_1 = e_2$.

Ein Typ T ist ein DT-Typ, wenn jeder geschlossene Wert vom Typ T nur aus Datenkonstruktoren aufgebaut ist. Z. B. sind $Bool$, $List\ Bool$ und Peano-Zahlen DT-Typen. Die Klasse der *DT-Formeln* umfasst genau jene geschlossene Formeln deren quantifizierte Variablen alle einen DT-Typ besitzen.

Formeln der Form $\forall x_1 :: T_n, \ldots, x_n :: T_n.\neg(e = \bot)$, wobei $\bot$ eine nichtterminierender geschlossener Ausdruck ist, drücken gerade aus, dass e ein terminierender Ausdruck ist. Solche Formeln werden als *TP-Formeln* bezeichnet.

Ein monomorph getypter, geschlossener $\mathcal{P}$-Ausdruck $e :: T_1 \to \ldots \to T_n \to T_{n+1}$ (wobei T_{n+1} kein Funktionstyp ist) ist *terminierend*, wenn $(e\ v_1\ \ldots\ v_n)\!\downarrow$ für alle geschlossenen $\mathcal{P}$-Werte $v_i :: T_i$ mit $i = 1, \ldots, n$ gilt.

Für eine Formel F seien die *Gleichungsausdrücke* von F, die Menge der Ausdrücke e_1, e_2, welche als linke und rechte Seiten von Gleichungen $e_1 = e_2$ in F vorkommen. Die Klasse der *ATD-Formeln* besteht aus allen geschlossenen $\mathcal{P}$-Formeln der Form

$$F := \forall x_1 :: T_1, \ldots, x_n :: T_n.F' \implies F'',$$

wobei F' und F'' quantorenfreie Formeln sind, alle Gleichungsausdrücke von F einen DT-Typ haben, alle Gleichungsausdrücke von F' terminierend sind und F' ist gerade eine Bedingung, die garantiert, dass alle Gleichungsausdrücke von F'' terminierend sind, d. h. für alle $\mathcal{P}$-Werte $v_1, \ldots, v_n$ gilt: Wenn $LMF(\mathcal{P}) \models F'[v_1/x_1, \ldots, v_n/x_n]$, dann sind alle Gleichungsausdrücke von $F''[v_1/x_1, \ldots, v_n/x_n]$ terminierend.

Z. B. ist die Formel $\forall x :: Nat.\neg(x = 0) \implies (div\ x\ x) = (S\ 0)$ eine ATD-Formel (wobei Nat der Typ für Peano-Zahlen ist und div ist die Division auf diesen Zahlen, wobei $div\ z\ 0 \sim \bot$ für alle Peano-Zahlen z.

In (Sabel & Schmidt-Schauß, 2013) wurde gezeigt, dass all diese Klassen konservativ bzgl. Programmerweiterungen sind:

Theorem 7.5.2. *Die Klassen der AEQ-, DT-, TP- und ATD-Formeln sind konservativ bezüglich Programmerweiterungen.*

Eine offene Frage ist, ob die Klassen der Formeln der Form $\forall x_1 : T_1, \ldots, x_n : T_n.A$ wobei A eine quantorenfreie Formel ist, konservativ bezüglich Programmerweiterungen ist. Allerdings umfassen die AEQ-,DT-,TP- und ATD-Formeln schon wesentliche Formeln, aus denen sich beispielsweise auch die Korrektheit von Induktionsschemata zur automatischen Deduktion der Tautologieeigenschaft für Formeln herleiten lässt (siehe (Sabel & Schmidt-Schauß, 2013)).

Schließlich wurde in (Sabel & Schmidt-Schauß, 2013) als Erweiterung auch kurz der Fall polymorph getypter Formeln betrachtet, wobei die Konservativität von Formelklassen im Allgemeinen nicht gilt. Selbst für AEQ-Formeln ist die Konservativität ein offenes Problem.

Insgesamt haben die Untersuchungen in (Sabel & Schmidt-Schauß, 2013) gezeigt, dass eine Programmlogik auf Basis der kontextuellen Äquivalenz aufgebaut werden kann, welche die „natürliche“ Behandlung von Nichtterminierung und partiellen Funktionen erlaubt. Dabei gelten die wesentlichen Theoreme für ausschließlich terminierende Funktionen weiterhin. Auf

Basis eines Satzes von korrekten Programmtransformationen (wie er z. B. in (Sabel & Schmidt-Schauß, 2013) für die Logik LMF zur Verfügung gestellt wird) können Theorembeweiser auch mit dieser Logik automatisiert Beweise erbringen. Die Integration dieses Ansatzes in bestehende Systeme erscheint möglich und erfordert anscheinend nur einige wenige Anpassungen. Als interessante Fragestellung lässt (Sabel & Schmidt-Schauß, 2013) offen, welche Eigenschaften (bzgl. der Konservativität) für polymorphe Formeln gelten.

8 Fazit

Die präsentierten Untersuchungen zeigen, dass auf Basis der formalen Semantik und insbesondere der auf der operationalen Semantik basierenden kontextuellen Gleichheit wesentliche Eigenschaften von Programmen und Programmiersprachen formal erfasst werden können. Insbesondere die Korrektheit von Programmtransformationen, die Korrektheit von Übersetzungen zwischen Programmkalkülen und die Integration der kontextuellen Gleichheit in eine Programmlogik sind hierbei zu nennen.

Die Ergebnisse zu konkreten Programmkalkülen – insbesondere zur Kernsprache von Haskell und deren nebenläufige Erweiterung – zeigen, dass auf Basis der allgemeinen Korrektheitsnotationen tatsächlich Korrektheiten und Eigenschaften für realistische und moderne Programmiersprachen formal nachgewiesen werden können.

Für zukünftige Arbeiten können zum einen die erarbeiteten Methoden und Techniken auf weitere Anwendungsfälle angewendet werden, um in der jeweiligen Programmiersprache Korrektheiten nachzuweisen. Zum anderen gibt es viele offene Fragestellungen, die in den einzelnen Kapiteln kurz diskutiert wurden, und daher weitere Forschung erfordern.

Literaturverzeichnis

Abramsky, S. (1990). The lazy lambda calculus. In **Turner, D. A.**, Hg., *Research topics in functional programming*, S. 65–116. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Alessi, F., Dezani-Ciancaglini, M. & de'Liguoro, U. (1994). May and must convergencey in concurrent lambda-calculus. In **Prívara, I., Rovan, B. & Ruzicka, P.**, Hg., *MFCS '94: Proceedings of the 19th International Symposium on Mathematical Foundations of Computer Science 1994*, Bd. 841 von *Lecture Notes in Comput. Sci.*, S. 211–220. Springer-Verlag, London, UK.

Antoy, S. & Hanus, M. (2010). Functional logic programming. *Commun. ACM*, 53(4):74–85.

Ariola, Z. M. & Felleisen, M. (1997). The call-by-need lambda calculus. *J. Funct. Programming*, 7(3):265–301.

Baader, F. & Nipkow, T. (1998). *Term Rewriting and All That*. Cambridge University Press.

Baker, H. C., Jr. & Hewitt, C. (1977). The incremental garbage collection of processes. In *Proceedings of the 1977 symposium on Artificial intelligence and programming languages*, S. 55–59. ACM, New York, NY, USA.

Barendregt, H. P. (1984). *The Lambda Calculus. Its Syntax and Semantics*. North-Holland, Amsterdam, New York.

Bertot, Y. & Castéran, P. (2004). *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag.

Bove, A., Krauss, A. & Sozeau, M. (2011). Partiality and recursion in interactive theorem provers: An overview. *Math. Structures Comput. Sci.* To appear, preprint available at `http://www4.informatik.tu-muenchen.de/~krauss/papers/recursion.pdf`.

Carayol, A., Hirschkoff, D. & Sangiorgi, D. (2005). On the representation of McCarthy's amb in the pi-calculus. *Theory Comput. Syst.*, 330(3):439–473.

Cheng, J. H. & Jones, C. B. (1991). On the usability of logics which handle partial functions. In **Morgan, C. & Woodcock, J. C. P.**, Hg., *Proceedings of the 3rd Refinement Workshop*, S. 51–69. Springer-Verlag.

Church, A. (1940). A formulation of the simple theory of types. *J. Symbolic Logic*, 5(2):56–68.

Church, A. (1941). *The Calculi of Lambda-Conversion*. Princeton University Press, Princeton, New Jersey.

Coquand, T. & Huet, G. P. (1988). The calculus of constructions. *Inform. and Comput.*, 76(2/3):95–120.

Damas, L. & Milner, R. (1982). Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '82, S. 207–212. ACM, New York, NY, USA.

de'Liguoro, U. & Piperno, A. (1992). Must preorder in non-deterministic untyped lambda-calculus. In **Raoult, J.-C.**, Hg., *CAAP '92: Proceedings of the 17th Colloquium on Trees in Algebra and Programming*, Bd. 581 von *Lecture Notes in Comput. Sci.*, S. 203–220. Springer-Verlag, London, UK.

Farmer, W. M. (1993). A simple type theory with partial functions and subtypes. *Ann. Pure Appl. Logic*, 64(3):211–240.

Farmer, W. M. (2004). Formalizing undefinedness arising in calculus. In **Basin, D. A. & Rusinowitch, M.**, Hg., *Automated Reasoning - Second International Joint Conference, IJCAR 2004, Cork, Ireland, July 4-8, 2004, Proceedings*, Bd. 3097 von *Lecture Notes in Comput. Sci.*, S. 475–489. Springer.

Felleisen, M. & Hieb, R. (1992). The revised report on the syntactic theories of sequential control and state. *Theory Comput. Syst.*, 103:235–271.

Ford, J. & Mason, I. A. (2001). Operational techniques in PVS - a preliminary evaluation. *Electron. Notes Theor. Comput. Sci.*, 42.

Ford, J. & Mason, I. A. (2003). Formal foundations of operational semantics. *Higher-Order Symbol. Comput.*, 16(3):161–202.

Fournet, C. & Gonthier, G. (1996). The reflexive cham and the join-calculus. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, S. 372–385. ACM, New York, NY, USA.

Fuhs, C., Giesl, J., Plücker, M., Schneider-Kamp, P. & Falke, S. (2009). Proving termination of integer term rewriting. In **Treinen, R.**, Hg., *RTA 20*, Bd. 5595 von *LNCS*, S. 32–47. Springer.

Giesl, J. (2001). Induction proofs with partial functions. *J. Automat. Reason.*, 26(1):1–49.

Giesl, J., Schneider-Kamp, P. & Thiemann, R. (2006). Aprove 1.2: automatic termination proofs in the dependency pair framework. In *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, Bd. 4130 von *Lecture Notes in Comput. Sci.*, S. 281–286. Springer-Verlag, Berlin, Heidelberg.

Goldberg, M. (2005). A variadic extension of Curry's fixed-point combinator. *Higher-Order Symbol. Comput.*, 18(3-4):371–388.

Gordon, A. D. (1994). A tutorial on co-induction and functional programming. In *Functional Programming, Glasgow 1994*, Workshops in Computing, S. 78–95. Springer-Verlag.

Gordon, A. D. (1999). Bisimilarity as a theory of functional programming. *Theory Comput. Syst.*, 228(1-2):5–47.

Halstead, R. H., Jr. (1985). Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7:501–538.

Hanus, M., Antoy, S., Braßel, B., Kuchen, H., López-Fraguas, F. J., Lux, W., Navarro, J. J. M. & Steiner, F. (2006). Curry – an integrated functional logic language. `http://www-ps.informatik.uni-kiel.de/currywiki/documentation/report`.

Hindley, R. (1969). The principal type-scheme of an object in combinatory logic. *Trans. Amer. Math. Soc.*, 146:29–60.

Howe, D. (1989). Equality in lazy computation systems. In *Proceedings of the Fourth Annual Symposium on Logic in computer science*, S. 198–203. IEEE Press, Piscataway, NJ, USA.

Howe, D. J. (1996). Proving congruence of bisimulation in functional programming languages. *Inform. and Comput.*, 124(2):103–112.

Hughes, G. E. & Cresswell, M. J. (1990). *Introduction to Modal Logic*. Routledge, London.

Kennaway, R., Klop, J. W., Sleep, M. R. & de Vries, F.-J. (1997). Infinitary lambda calculus. *Theory Comput. Syst.*, 175(1):93–125.

Kutzner, A. (2000). *Ein nichtdeterministischer call-by-need Lambda-Kalkül mit erratic choice: Operationale Semantik, Programmtransformationen und Anwendungen*. Dissertation, J.W.Goethe-Universität Frankfurt.

Kutzner, A. & Schmidt-Schauß, M. (1998). A nondeterministic call-by-need lambda calculus. In *International Conference on Functional Programming 1998*, S. 324–335. ACM Press.

Laneve, C. (1996). On testing equivalence: May and must testing in the join-calculus. Techn. Ber. Technical Report UBLCS 96-04, University of Bologna.

Lassen, S. B. (1998). *Relational Reasoning about Functions and Nondeterminism*. Dissertation, Department of Computer Science, University of Aarhus. BRICS Dissertation Series DS-98-2.

Lassen, S. B. & Pitcher, C. S. (2000). Similarity and bisimilarity for countable non-determinism and higher-order functions. *Electron. Notes Theor. Comput. Sci.*, 10.

Levy, P. B. (2007). Amb breaks well-pointedness, ground amb doesn't. *Electron. Notes Theor. Comput. Sci.*, 173(1):221–239.

Mann, M. (2005a). *A Non-Deterministic Call-by-Need Lambda Calculus: Proving Similarity a Precongruence by an Extension of Howe's Method to Sharing*. Dissertation, J. W. Goethe-Universität, Frankfurt.

Mann, M. (2005b). Congruence of bisimulation in a non-deterministic call-by-need lambda calculus. *Electron. Notes Theor. Comput. Sci.*, 128(1):81–101.

Mann, M. & Schmidt-Schauß, M. (2010). Similarity implies equivalence in a class of non-deterministic call-by-need lambda calculi. *Inform. and Comput.*, 208(3):276 – 291.

Maraist, J., Odersky, M. & Wadler, P. (1998). The call-by-need lambda calculus. *J. Funct. Programming*, 8(3):275–317.

Mason, I., Smith, S. F. & Talcott, C. L. (1996). From operational semantics to domain theory. *Inform. and Comput.*, 128:26–47.

Mason, I. & Talcott, C. L. (1991). Equivalence in functional languages with effects. *J. Funct. Programming*, 1(3):287–327.

Milner, R. (1977). Fully abstract models of typed lambda calculi. *Theory Comput. Syst.*, 4(1):1–22.

Milner, R. (1978). A theory of type polymorphism in programming. *J. Comput. System Sci.*, 17(3):348–375.

Milner, R. (1999). *Communicating and mobile systems: the π-calculus*. Cambridge University Press, New York, NY, USA.

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

Moran, A., Sands, D. & Carlsson, M. (2003). Erratic fudgets: a semantic theory for an embedded coordination language. *Sci. Comput. Programming*, 46(1-2):99–135.

Moran, A. K. D., Sands, D. & Carlsson, M. (1999). Erratic fudgets: A semantic theory for an embedded coordination language. In *Coordination '99*, Bd. 1594 von *Lecture Notes in Comput. Sci.*, S. 85–102. Springer-Verlag.

Morris, J. H., Jr. (1968). *Lambda Calculus Models of Programming Languages*. Dissertation, MIT, Cambridge, MA.

Niehren, J., Sabel, D., Schmidt-Schauß, M. & Schwinghammer, J. (2007). Observational semantics for a concurrent lambda calculus with reference cells and futures. *Electron. Notes Theor. Comput. Sci.*, 173:313–337.

Niehren, J., Schwinghammer, J. & Smolka, G. (2006). A concurrent lambda calculus with futures. *Theory Comput. Syst.*, 364(3):338–356.

Nipkow, T., Paulson, L. C. & Wenzel, M. (2002). *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, Bd. 2283 von *LNCS*. Springer.

Ong, C.-H. L. (1993). Non-determinism in a functional setting. In *Proc. 8th IEEE Symposium on Logic in Computer Science (LICS '93)*, S. 275–286. IEEE Computer Society Press.

Peyton Jones, S. (2001). Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In **Tony Hoare, R. S., Manfred Broy**, Hg., *Engineering theories of software construction*, S. 47–96. IOS-Press. Presented at the 2000 Marktoberdorf Summer School.

Peyton Jones, S., Hg. (2003). *Haskell 98 language and libraries: the Revised Report*. Cambridge University Press. `www.haskell.org`.

Peyton Jones, S., Gordon, A. & Finne, S. (1996). Concurrent Haskell. In *Proc. 23th ACM Principles of Programming Languages*, S. 295–308. ACM.

Peyton Jones, S. & Singh, S. (2009). A tutorial on parallel and concurrent programming in haskell. In *Proceedings of the 6th international conference on Advanced functional programming*, AFP'08, S. 267–305. Springer-Verlag, Berlin, Heidelberg.

Peyton Jones, S. L. & Santos, A. L. M. (1994). Compilation by transformation in the Glasgow Haskell compiler. In **Hammond, K., Turner, D. N. & Sansom, P. M.**, Hg., *Functional Programming*, S. 184–204. Springer, Berlin, Heidelberg.

Peyton Jones, S. L. & Wadler, P. (1993). Imperative functional programming. In *Proceedings 20th Symposium on Principles of Programming Languages, Charleston, South Carolina,*, S. 71–84. ACM.

Pitts, A. M. (2011). Howe's method for higher-order languages. In **Sangiorgi, D. & Rutten, J.**, Hg., *Advanced Topics in Bisimulation and Coinduction*, Bd. 52 von *Cambridge Tracts in Theoretical Computer Science*, Kap. 5, S. 197–232. Cambridge University Press.

Pitts, A. M. & Stark, I. D. B. (1998). Operational reasoning for functions with local state. In **Gordon, A. D. & Pitts, A. M.**, Hg., *Higher Order Operational Techniques in Semantics*, Publications of the Newton Institute, S. 227–273. Cambridge University Press.

Plotkin, G. D. (1975). Call-by-name, call-by-value, and the lambda-calculus. *Theory Comput. Syst.*, 1:125–159.

Rau, C., Sabel, D. & Schmidt-Schauß, M. (2012). Correctness of program transformations as a termination problem. In **Gramlich, B., Miller, D. & Sattler, U.**, Hg., *Automated Reasoning – Proceedings of the 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012*, Bd. 7364 von *Lecture Notes in Comput. Sci.*, S. 462–476. Springer Berlin / Heidelberg.

Rau, C. & Schmidt-Schauß, M. (2010). Towards correctness of program transformations through unification and critical pair computation. In *Proceedings of the 24th International Workshop on Unification*, Bd. 42 von *Electronic Proceedings in Theoretical Computer Science*, S. 39–54.

Rau, C. & Schmidt-Schauß, M. (2011). A unification algorithm to compute overlaps in a call-by-need lambda-calculus with variable-binding chains. In *Proceedings of the 25th International Workshop on Unification*, S. 35–41.

Rensink, A. & Vogler, W. (2007). Fair testing. *Inform. and Comput.*, 205(2):125–198.

Rossberg, A., Botlan, D. L., Tack, G., Brunklaus, T. & Smolka, G. (2006). *Alice Through the Looking Glass*, Bd. 5 von *Trends in Functional Programming*, S. 79–96. Intellect Books, Bristol, UK, Munich, Germany.

Sabel, D. (2008). *Semantics of a Call-by-Need Lambda Calculus with McCarthy's amb for Program Equivalence*. Dissertation, J. W. Goethe-Universität Frankfurt, Institut für Informatik. Fachbereich Informatik und Mathematik.

Sabel, D. (2012a). An abstract machine for Concurrent Haskell with futures. Frank report 48, Institut für Informatik, Goethe-Universität Frankfurt am Main.

Sabel, D. (2012b). An abstract machine for concurrent haskell with futures. In **Jähnichen, S., Rumpe, B. & Schlingloff, H.**, Hg., *Software Engineering 2012 - Workshopband, Fachtagung des GI-Fachbereichs Softwaretechnik, 27. Februar - 2. März 2012 in Berlin (5. Arbeitstagung Programmiersprache, ATPS 2012)*, Bd. 199 von *GI Edition - Lecture Notes in Informatics*, S. 29–44. Köllen Druck+Verlag.

Sabel, D. & Schmidt-Schauß, M. (2008). A call-by-need lambda-calculus with locally bottom-avoiding choice: Context lemma and correctness of transformations. *Math. Structures Comput. Sci.*, 18(03):501–553.

Sabel, D. & Schmidt-Schauß, M. (2011a). A contextual semantics for concurrent Haskell with futures. Frank report 44, Institut für Informatik. Fachbereich Informatik und Mathematik. J. W. Goethe-Universität Frankfurt am Main.

Sabel, D. & Schmidt-Schauß, M. (2011b). A contextual semantics for Concurrent Haskell with futures. In *Proceedings of the 13th international ACM SIGPLAN symposium on Principles and practices of declarative programming*, PPDP '11, S. 101–112. ACM, New York, NY, USA.

Sabel, D. & Schmidt-Schauß, M. (2011c). On conservativity of Concurrent Haskell. Frank report 47, Institut für Informatik, Goethe-Universität Frankfurt am Main.

Sabel, D. & Schmidt-Schauß, M. (2011d). Reconstruction of a logic for inductive proofs of properties of functional programs. Frank report 39, Institut für Informatik. Fachbereich Informatik und Mathematik. J. W. Goethe-Universität Frankfurt am Main.

Sabel, D. & Schmidt-Schauß, M. (2012). Conservative concurrency in Haskell. In **Dershowitz, N.**, Hg., *Proceedings of the 27th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2012), 25-28 June 2012, Dubrovnik, Croatia*, S. 561–570.

Sabel, D. & Schmidt-Schauß, M. (2013). A two-valued logic for properties of strict functional programs allowing partial functions. *Journal of Automated Reasoning*, 50(4):383–421. ISSN 0168-7433.

Sabel, D., Schmidt-Schauß, M. & Harwath, F. (2009). Reasoning about contextual equivalence: From untyped to polymorphically typed calculi. In **Fischer, S., Maehle, E. & Reischuk, R.**, Hg., *INFORMATIK 2009, Im Focus das Leben, Beiträge der 39. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 28.9 - 2.10.2009 in Lübeck*, Bd. 154 von *GI Edition - Lecture Notes in Informatics*, S. 369; 2931–45. Köllen Druck+Verlag. (4. Arbeitstagung Programmiersprachen (ATPS)).

Sangiorgi, D. & Walker, D. (2001). *The π-Calculus: A Theory of Mobile Processes*. Cambridge University Press, New York, NY, USA.

Schieder, B. & Broy, M. (1999). Adapting calculational logic to the undefined. *Comput. J.*, 42(2):73–81.

Schlosser, A., Walther, C., Gonder, M. & Aderhold, M. (2007). Context dependent procedures and computed types in verifun. *Electron. Notes Theor. Comput. Sci.*, 174(7):61–78.

Schmidt-Schauß, M. (2003). FUNDIO: A Lambda-Calculus with a `letrec`, `case`, Constructors, and an IO-Interface: Approaching a Theory of `unsafePerformIO`. Frank report 16, Institut für Informatik, J.W. Goethe-Universität Frankfurt.

Schmidt-Schauß, M. (2007). Correctness of copy in calculi with letrec. In *Term Rewriting and Applications (RTA-18)*, Bd. 4533 von *Lecture Notes in Comput. Sci.*, S. 329–343. Springer.

Schmidt-Schauß, M. & Machkasova, E. (2008). A finite simulation method in a non-deterministic call-by-need calculus with letrec, constructors and case. In *Proc. of RTA 2008*, Nr. 5117 in LNCS, S. 321–335. Springer-Verlag.

Schmidt-Schauß, M., Machkasova, E. & Sabel, D. (2009a). Counterexamples to simulation in non-deterministic call-by-need lambda-calculi with letrec. Frank report 38, Institut für Informatik. Fachbereich Informatik und Mathematik. J. W. Goethe-Universität Frankfurt am Main.

Schmidt-Schauß, M., Niehren, J., Schwinghammer, J. & Sabel, D. (2008a). Adequacy of compositional translations for observational semantics. Frank report 33, Institut für Informatik. Fachbereich Informatik und Mathematik. J. W. Goethe-Universität Frankfurt am Main.

Schmidt-Schauß, M., Niehren, J., Schwinghammer, J. & Sabel, D. (2008b). Adequacy of compositional translations for observational semantics. In **Ausiello, G., Karhumäki, J., Mauri,**

G. & Ong, C.-H. L., Hg., *Fifth IFIP International Conference On Theoretical Computer Science - TCS 2008, IFIP 20th World Computer Congress, TC 1, Foundations of Computer Science, September 7-10, 2008, Milano, Italy*, Bd. 273 von *IFIP*, S. 521–535. Springer.

Schmidt-Schauß, M. & Sabel, D. (2007). Program transformation for functional circuit descriptions. Frank report 30, Institut für Informatik. Fachbereich Informatik und Mathematik. J. W. Goethe-Universität Frankfurt am Main.

Schmidt-Schauß, M. & Sabel, D. (2008). Closures of may and must convergence for contextual equivalence. Frank report 35, Institut für Informatik. Fachbereich Informatik und Mathematik. J. W. Goethe-Universität Frankfurt am Main.

Schmidt-Schauß, M. & Sabel, D. (2010a). Closures of may-, should- and must-convergences for contextual equivalence. *Inform. Process. Lett.*, 110(6):232 – 235.

Schmidt-Schauß, M. & Sabel, D. (2010b). On generic context lemmas for higher-order calculi with sharing. *Theory Comput. Syst.*, 411(11-13):1521 – 1541.

Schmidt-Schauß, M., Sabel, D. & Harwath, F. (2009b). Contextual equivalence in lambda-calculi extended with letrec and with a parametric polymorphic type system. Frank report 36, Institut für Informatik. Fachbereich Informatik und Mathematik. J. W. Goethe-Universität Frankfurt am Main.

Schmidt-Schauß, M., Sabel, D. & Machkasova, E. (2010a). Simulation in the call-by-need lambda-calculus with letrec. Frank report 40, Institut für Informatik. Fachbereich Informatik und Mathematik. J. W. Goethe-Universität Frankfurt am Main.

Schmidt-Schauß, M., Sabel, D. & Machkasova, E. (2010b). Simulation in the call-by-need lambda-calculus with letrec. In **Lynch, C.**, Hg., *Proceedings of the 21st International Conference on Rewriting Techniques and Applications*, Bd. 6 von *Leibniz International Proceedings in Informatics (LIPIcs)*, S. 295–310. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany.

Schmidt-Schauß, M., Sabel, D. & Machkasova, E. (2011). Counterexamples to applicative simulation and extensionality in non-deterministic call-by-need lambda-calculi with letrec. *Inform. Process. Lett.*, 111(14):711–716.

Schmidt-Schauß, M., Sabel, D. & Machkasova, E. (2012). Simulation in the call-by-need lambda-calculus with letrec, case, constructors, and seq. Frank report 49, Institut für Informatik. Fachbereich Informatik und Mathematik. J. W. Goethe-Universität Frankfurt am Main.

Schmidt-Schauß, M., Schütz, M. & Sabel, D. (2008c). Safety of Nöcker's strictness analysis. *J. Funct. Programming*, 18(04):503–551.

Schwinghammer, J., Sabel, D., Niehren, J. & Schmidt-Schauß, M. (2009a). On correctness of buffer implementations in a concurrent lambda calculus with futures. Frank report 37, Institut für Informatik. Fachbereich Informatik und Mathematik. Goethe-Universität Frankfurt am Main.

Schwinghammer, J., Sabel, D., Schmidt-Schauß, M. & Niehren, J. (2009b). Correctly translating concurrency primitives. In *ML '09: Proceedings of the 2009 ACM SIGPLAN workshop on ML*, S. 27–38. ACM, New York, NY, USA.

Sestoft, P. (1997). Deriving a lazy abstract machine. *J. Funct. Programming*, 7(3):231–264.

Wadler, P. (1995). Monads for functional programming. In **Jeuring, J. & Meijer, E.**, Hg., *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text*, Bd. 925 von *Lecture Notes in Comput. Sci.*, S. 24–52. Springer.

Walther, C. (1994). Mathematical induction. In **Gabbay, D. M., Hogger, C. J., Robinson, J. A. & Siekmann, J. H.**, Hg., *Handbook of Logic in Artificial Intelligence and Logic Programming*, Bd. 2, S. 127–228. Oxford University Press.

Walther, C. & Schweitzer, S. (2005). Reasoning about incompletely defined programs. In *12. LPAR '05*, LNCS 3835, S. 427–442.

Abbildungsverzeichnis