FRIEDRICH-SCHILLER-UNIVERSITÄT JENA

JENAER SCHRIFTEN

ZUR

MATHEMATIK UND INFORMATIK

Eingang: 22.08.2017 Math/Inf/02/2017 Als Manuskript gedruckt

Wolfram Amme, Thomas S. Heinze (Hrsg.)

Programmiersprachen und Grundlagen der Programmierung
19. Kolloquium, KPS 2017
Weimar, 25.–27. September 2017
Tagungsband

Wolfram Amme, Thomas S. Heinze (Hrsg.)

Programmiersprachen und Grundlagen der Programmierung

19. Kolloquium, KPS 2017
Weimar, 25.–27. September 2017
Tagungsband

Wolfram Amme
Friedrich-Schiller-Universität Jena, Institut für Informatik
Ernst-Abbe-Platz 2, 07743 Jena
wolfram.ammme@uni-jena.de

Thomas S. Heinze
Friedrich-Schiller-Universität Jena, Institut für Informatik
Ernst-Abbe-Platz 2, 07743 Jena
t.heinze@uni-jena.de

Bericht Math/Inf/02/2017
Jenaer Schriften zur Mathematik und Informatik
Friedrich-Schiller-Universität Jena

Vorwort

Das 19. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS 2017) setzt eine Reihe von Arbeitstagungen fort, die ursprünglich von den Forschungsgruppen der Professoren Friedrich L. Bauer (TU München), Klaus Indermark (RWTH Aachen) und Hans Langmaack (CAU Kiel) ins Leben gerufen wurde. Das erste Kolloquium fand 1980 in Tannenfelde im Naturpark Aukrug in der Nähe von Neumünster in Schleswig-Holstein statt. Die Tagungen finden seitdem in etwa zweijährlichem Rhythmus statt. Aus den ursprünglich drei Arbeitsgruppen sind in der Zwischenzeit weitere Forschungsgruppen in ganz Deutschland und darüber hinaus hervorgegangen. Heute präsentiert sich das Kolloquium als offenes Forum für alle interessierten deutschsprachigen Wissenschaftler und bietet einen zwanglosen Rahmen zum Austausch neuer Ideen und Ergebnisse aus den Bereichen Entwurf und Implementierung von Programmiersprachen sowie Grundlagen und Methodik des Programmierens.

Die inzwischen 37-jährige Tradition wird sichtbar in der Liste der bisherigen Tagungsorte und Veranstalter:

2015	Pörtschach am Wörthersee	TU Wien
2013	Lutherstadt Wittenberg	Uni Halle-Wittenberg
2011	Schloss Raesfeld	Uni Münster
2009	Maria Taferl	TU Wien
2007	Timmendorfer Strand	Uni Lübeck
2005	Fischbachau	LMU München
2004	Freiburg-Munzingen	Uni Freiburg
2001	Rurberg in der Eifel	RWTH Aachen
1999	Kirchhundem-Heinsberg	FernUni Hagen
1997	Avendorf auf Fehmarn	Uni Kiel
1995	Alt-Reichenau	Uni Passau
1993	Garmisch-Partenkirchen	UniBw München
1992	Rothenberge bei Steinfurt	Uni Münster
1989	Hirschegg	Uni Augsburg
1987	Midlum auf Föhr	Uni Kiel
1985	Passau	Uni Passau
1982	Altenahr	RWTH Aachen
1980	Tannenfelde im Naturpark Aukrug	Uni Kiel

Das 19. Kolloquium Programmiersprachen und Grundlagen der Programmierung wird vom Fachbereich Programmiersprachen und Übersetzerbau am Institut für Informatik der Friedrich-Schiller-Universität Jena organisiert. Wir freuen uns darauf, 30 Teilnehmer vom 25. bis 27. September 2017 in Weimar begrüßen zu dürfen, darunter Prof. Dr. Dr. h.c. Hans Langmaack, einen der Gründungsväter der Veranstaltungsreihe, und Prof. Dr. Dr. h.c. Gerhard Goos mit einem Beitrag zur Geschichte der deutschsprachigen Informatik im Bereich Programmiersprachen und Übersetzerbau.

Der vorliegende Tagungsband ist als Bericht Math/Inf/02/2017 in den Jenaer Schriften zur Mathematik und Informatik der Fakultät für Mathematik und Informatik der Friedrich-Schiller-Universität Jena erschienen. Die darin enthaltenen 22 Beiträge zeigen, zum Teil in Form von Kurzzusammenfassungen, die Breite der wissenschaftlichen Forschung zu Programmiersprachen und Grundlagen der Programmierung im deutschsprachigen Raum. Unser ausdrücklicher Dank gilt allen Autoren für ihre Beträge, ohne die dieser Tagungsband natürlich nicht möglich wäre. Unser Dank gilt darüber hinaus Cornelia Müsse, der Sekretärin unserer Arbeitsgruppe, sowie den Mitarbeitern des Hauses Dorint Am Goethepark für die gute Unterstützung bei der Planung und Vorbereitung dieses Treffens.

Wir freuen uns auf interessante und spannende Vorträge und möchten allen Teilnehmern fruchtbare Diskussionen und vielfältige Anregungen für die eigenen Forschungsarbeiten, das Kennenlernen von Kollegen und das Wiedersehen von guten Bekannten, das Anbahnen neuer und die Vertiefung bestehender Kooperationen, sowie nicht zuletzt einen angenehmen und hoffentlich stimulierenden Aufenthalt in der Kulturstadt Weimar wünschen.

Jena, August 2017

Wolfram Amme
Thomas S. Heinze

Inhaltsverzeichnis

Geschichte der deutschsprachigen Informatik Programmiersprachen und Übersetzerbau

Gerhard Goos
Fakultät für Informatik
KIT

7. August 2017

Zusammenfassung

Zusammenfassung: Kurzdarstellung wichtiger Entdeckungen, Entwicklungen und Erfindungen der deutschsprachigen Informatik im Bereich Programmiersprachen und Übersetzerbau.

1 Der Beginn

Den Beginn der modernen Informatik im deutshsprachigen Raum bilden die Unentscheidbarkeitssätze von Kurt Gödel, (Gödel, 1931), also ein Ergebnis der Logik.

Den praktischen Beginn der modernen Informatik im deutschsprachigen Raum markieren die Arbeiten von Konrad Zuse ab 1936. Dazu zählt seine Entscheidung das Binärsystem zu benutzen, vor allem aber die Verwendung der binären Gleitkommadarstellung für die Verarbeitung von Zahlen in der Relaismaschine Z1. Zwar waren halblogarithmische Zahldarstellungen der Form $m \times 60^e$ bereits den Sumerern vor 4700 Jahren bekannt und die Mathematiker und Naturwissenschaftler benutzen seit langem dezimal $m \times 10^e$. Aber die Zahldarstellung im Binärsystem $m \times 2^e$ mit normalisiertem $m = 1, m_1 m_2 \ldots$ war neu. Auch, daß die führende Ziffer $m_0 = 1$ wie auch im heutigen Standard (IEEE 754, 2008) nicht explizit gespeichert wird, findet sich bereits bei Zuse. Die allgemeine Entwicklung von Rechenanlagen übernahm die Gleitkommaarithmetik erst in den 50er Jahren. Die IBM 704 war 1954 der erste weit verbreitete Rechner mit Gleitkommaoperationen in Hardware. Zuvor arbeitete man mit Festkommaarithmetik, bei der die Lage des Dezimalkommas mühsam getrennt festgelegt wurde. Beim Umgang mit Geld ist die Art der dezimalen Rundung gesetzlich festgeschrieben und mit binärer Arithmetik nicht einfach nachvollziehbar.

Die zweite große Leistung Konrad Zuses im Bereich Programmiersprachen war die Erfindung des *Plankalküls* 1944/45, der ersten höheren Programmiersprache, (Zuse, 1972; Bauer und Wössner, 1972). Zwar wurde dies erst 1972 veröffentlicht und die zweidimensionale Scchreibweise fand bisher keine Nachahmer. Aber es handelt sich um die erste **goto**-freie Programmiersprache mit Datentypen, die sich anderswo erst im Laufe der 60er

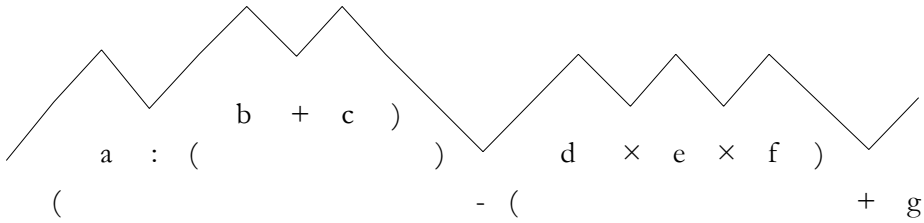


Abbildung 1: Rutishausers Klammergebirge

Jahre etablierten. Konrad Zuse war übrigens auch ein begnadeter Maler, wie viele Bilder z. B. aus seiner Zeit in Hopferau/Allgäu am Ende des zweiten Weltkriegs zeigen. []

2 Die 50er Jahre

(Wilkes et al., 1951) hatten 1951 das erste Programmierhandbuch für das Programmieren in Maschinensprache auf der EDVAC veröffentlicht. Zugleich begann die Suche nach Verarbeitungsmöglichkeiten für arithmetische Ausdrücke, die in gewöhnlicher mathematischer Schreibweise dargestellt waren.

Die erste Lösung hierfür fand Heinz Rutishauser in seiner Habilitationsschrift, (Rutishauser, 1951), mit dem sogenannten Klammergebirge. Rutishauser implementierte diese Lösung auch auf der ERMETH, dem unter Leitung von Prof. Stiefel gebauten Rechner der ETH Zürich. Das Verfahren hatte allerdings quadratischen Aufwand in der Anzahl der Symbole einer Formel. Später und bis heute benutzt man die umgekehrte polnische Normalform oder Postfixform von Jan Łukasiewicz (1878-1956), mit der sich die Berechnung mit linearem Aufwand erledigen läßt.

Rutishauser war auch eine der treibenden Kräfte bei der Definition von ALGOL 58 und ALGOL 60.

Die eigentliche Arbeit an höheren Programmiersprachen und dem Bau von Übersetzern begann im deutschsprachigen Raum 1955 mit den Herren Heinz Rutishauser, Klaus Samelson und Friedrich L. Bauer. Dazu stieß etwas später der Darmstädter Hermann Bottenbruch. Ihr Ziel war eine einheitliche, maschinenunabhängige Programmiersprache, vgl. (Bauer, 1994).

Dazu formulierte Herr Samelson bereits 1955, (Samelson, 1957), die Grundzüge des Kellerprinzips und zwar nicht nur für die Berechnung arithmetischer Ausdrücke, sondern auch für offene und geschlossene Unterprogramme, also für Programmblöcke. Die Grundidee des Verfahrens geht auf Helmut Angstl zurück, der bereits 1950 in einem Vortrag im Logik-Seminar von Prof. Britzelmayer an der LMU eine Vorstufe des Kellerprinzips zur Überprüfung der Wohlgeformtheit einer aussagenlogischen Formel in Präfixform vorgestellt hatte, damals als ein mechanisches Gerät. Seine Idee wurde dann bis 1957 als Relais-Rechner zur Berechnung aussagenlogischer Formeln realisiert, (Bauer, 1960; Bauer, 1977). Auf Anraten des Münchner Elektroingenieurs Hans Piloty, der das Prinzip für einen Rechner PERM2 nutzen wollte, wurde das Verfahren, übertragen auf arithmetische Ausdrücke, zunächst als Patentschrift veröffentlicht, (Bauer und Samelson, 1957). Bauer und Samelson konstruierten dazu eine Rechenmaschine, die einen Operanden- und einen

Operationskeller eingebaut hatte. Dafür erhielten sie ein deutsches Patent und später auch Patente in anderen Ländern, darunter den USA. Die Idee getrennter Operanden- und Operationskeller lag später auch vielen Implementierungen funktionaler Programmiersprachen zugrunde.

Ab 1957/58 verfolgten Bauer, Bottenbruch, Rutishauser und Samelson dann den Gedanken einer einheitlichen „algebraischen" Programmiersprache, um Algorithmen unabhängig von den Eigenheiten von Rechnern beschreiben zu können. Sie warben dafür um Mitstreiter. Aus den USA kooperierten John Backus, Charles Katz, Alan Perlis und Joseph Henry Wegstein. Das Ergebnis der Arbeit nannten die Amerikaner IAL, *international algebraic language*, in Europa hieß die Sprache ALGOL 58, (Perlis und Samelson, K. (Hrsg.), 1958, 1959). Der Begriff *algorithmic language* und die Abkürzung ALGOL stammen der Überlieferung nach von Herrn Bottenbruch. Die Sprache konnte sich nicht durchsetzen, führte aber in Amerika zu zwei Dialekten Jovial und Neliac, die bis zur Einführung von Ada die Echtzeitprogrammierung vor allem im militärischen Bereich dominierten. Indirekt stammt nach einigen Zwischenschritten auch die Programmiersprache C von Algol 58 ab.

Das Kellerprinzip führte K. Samelson umstandslos zur Erfindung des Codeblocks **begin . . . end** und damit des Prinzips der Blockschachtelung in Algol 58. Damit waren alle Elemente vorhanden, die zur Definition von Algol 60, benötigt wurden.

3 Die 60er Jahre

Algol 60 wurde auf einer Konferenz in Paris im Januar 1960 mit den Teilnehmern Friedrich L. Bauer, Peter Naur, Heinz Rutishauser, Klaus Samelson, Bernard Vauquois, Adriaan van Wijngaarden, Michael Woodger aus Europa und John W. Backus, Julien Green, Charles Katz, John McCarthy, Alan J. Perlis, Joseph Henry Wegstein aus den USA definiert. Peter Naur war der Herausgeber des Berichts, der als (Naur P. (Hrsg.), 1960) veröffentlicht wurde. Die noch heute gültige Sprachnorm ist die Revision (Naur P. (Hrsg.), 1962). Das Urteil von Sir C.A.R. Hoare lautete „Here is a language so far ahead of its time that it was not only an improvement on its predecessors but also on nearly all its successors", Hoare (1973).

Die Teilnehmer der Pariser Konferenz bildeten hernach den Kern der Working Group 2.1 der International Federation for Information Processing, die 1962 gegründet wurde und die weitere Entwicklung von Algol betreute.

Der Sprache fehlte eine Definition der Ein/Ausgabe. Auch verstand sich Algol 60 ausschließlich als Sprache zur Beschreibung numerischer Algorithmen. Kommerzielle Anwendungen, die naturgemäß auch den Umgang mit Geldbeträgen und der gesetzlich vorgeschriebenen dezimalen Rundung vorausgesetzt hätten, sowie Anwendungen zur Manipulation von Texten waren nicht vorgesehen.

Rekursive Prozeduren und Funktionen wurden in Algol 60 erst in letzter Minute auf Wunsch von van Wijngaarden und Edsgar W. Dijkstra zugelassen. Die deutschen Mitglieder lehnten das ab und die ab Juni 1959 aus der ZMMD-Kooperation (Zürich, München, Mainz, Darmstadt) entstandene ALCOR-Gruppe implementierte Algol überwiegend ohne Rekursion, (Baumann, 1961). Die ersten Übersetzer in Deutschland und Österreich wurden für die Zuse Z22 (M. Paul), PERM (G. Seegmüller), ERMETH

(H. R. Schwarz), Mailüfterl (P. Lucas, H. Bekic), Zebra (W. L. van der Poel, van der Mey) , Siemens 2002 (U. Hill-Samelson, H. Langmaack) Ende 1961/Anfang 1962 nach einheitlichen Bauplänen fertiggestellt. Die Methodik für die syntaktische Analyse war eine Verallgemeinerung des Kellerprinzips von Samelson und Bauer. Im Anfang gab es noch keine statische semantische Analyse, sondern es wurde eine weitgehend maschinenunabhängige Zwischensprache erzeugt, die Datentypen wie real und integer interpretativ zur Laufzeit unterschied. Dies war vor allem auch durch die Speicherbeschränkungen verursacht: Keine der Maschinen hatte im Anfang einen Hintergrundspeicher brauchbarer Größe, um ein übersetztes Programm dauerhaft zu speichern. Das Programm mußte also unmittelbar nach der Übersetzung ausgeführt werden.

Die Grundzüge dieser Vorgehensweise wurden in Samelson und Bauer (1959) veröffentlicht. Der Artikel war so bedeutsam, daß ihn die Communications ACM, (Samelson und Bauer, 1960), in englischer Übersetzung nachdruckten. Er war für viele Jahre einer der internationalen Eckpfeiler des Übersetzerbaus.

Ab 1962 begann eine zweite Phase der ALCOR-Kooperation. Die Rechner waren etwas schneller und größer geworden und man konnte nun bequem volles Algol 60 effizient implementieren. Dies führte zu Seegmüllers Algol 60-Übersetzer für den Telefunken TR4 von 1962 und dem Übersetzer für die Siemens 2002, die bereits lineare Adreßfortschaltung, den praktisch wichtigsten Aspekt von *strength reduction* also einer Optimierung enthielten, Hill et al. (1962); Grau et al. (1967).

Eine weitere, lange Zeit wenig beachtete Leistung war die Erfindung des *post mortem Speicherabzugs*, (Seegmüller, 1962; Bayer et al., 1967). Er lieferte nach einem Programmabsturz eine komplette Darstellung des Programmspeichers, gegliedert entsprechend den gerade aufgerufenen Prozeduren, mit den Namen der vorhandenen Variablen und den Variablenwerten gemäß dem Variablentyp; das galt auch für Reihungen. In einer Zeit, wo man oft einen Tag warten mußte, um das Ergebnis eines Programmlaufs vom Rechenzentrum zu erhalten, war dies eine bedeutende Erleichterung beim Testen. Ein österreichischer Kollege schrieb damals, daß Maschinensprache besser sei als die Benutzung von Cobol, da man dann wenigstens den (sedezimalen) Speicherabzug nachvollziehen könne. Auch heute noch wäre ein gegliederter Speicherabzug oft besser als die Benutzung des gdb.

Die Arbeit (Eickel et al., 1963) faßt die Grundlagen des Syntaxanalyseverfahrens für kontextfreie Sprachen mit beschränktem Kontext zusammen, das die ALCOR-Gruppe verwendete. Ausgangspunkt war die Dissertation (Paul, 1962). (m, k)-beschränkte Grammatiken sind eine echte Teilmenge der $LR(k)$-Grammatiken. Im Vergleich sieht man für $k = 1$ keine wesentlichen Unterschiede zu den später gebräuchlichen LALR(1)-Analysatoren. Daß man damit sämtliche deterministische kontextfreie Grammatiken erfassen kann, wurde erst später erkannt.

F.L. Bauer erklärte 1965 die Forschung im Bereich Übersetzerbau für beendigt, das Weitere sei Sache der Industrie. Das war zweifellos eine Fehlprognose, deren Ursache die damals weit verbreitete Meinung war, daß semantische Analyse, Codeoptimierung und Codeerzeugung ganz einfache Angelegenheiten seien, die keiner weiteren Aufarbeitung bedürfen. Dies wurde danach sehr schnell widerlegt, z.B. durch die Typenvielfalt der ersten objektorientierten Programmiersprache Simula 67, (Dahl et al., 1968), und die zunehmende Komplexität der Maschinensprachen.

Wie wenig Industrie und Hochschulen in Deutschland in den 60er Jahren zusammenwirkten, konnte man 1967 in München erleben, als IBM Software-Spezialisten schickte, die den Mitarbeitern der TU die Vorzüge von

Algol 60 und die Verfügbarkeit dieser Sprache auf der IBM 360/81 darstellen sollten. Die Herren ahnten nicht, daß sie wesentliche Erfinder dieser Sprache vor sich sitzen hatten. Das Verhältnis änderte sich erst in den 70er Jahren.

Mit der Definition von LISP hatte J. McCarthy, (McCarthy, 1960), die Idee geboren, die Semantik von Programmiersprachen durch einen formalen Interpretierer zu beschreiben. Heinz Zemanek, der Konstrukteur des Rechners Mailüfterl an der TU Wien, wechselte nach seiner Assistentenzeit zur Firma IBM, übernahm die Leitung des neu gegründeten IBM-Labors Wien und griff den Gedanken auf, um zusammen mit einer großen Zahl von Mitarbeitern eine formale Definition der Programmiersprache PL/1 zu erreichen. Der Grundgedanke dieser als *Vienna Definition Language* (VDL) bekannten Methode besteht darin, sowohl ein Programm als auch seine Daten als Bäume aufzufassen. Die wesentliche Operation lautet $\mu(x; s : y)$, sie liefert das Objekt x, in dem das durch den Selektor s ausgewählte Teilobjekt durch y ersetzt ist. Gibt es in x keinen Selektor s, so wird $s : y$ neu hinzugefügt, bei $y = \Omega$ wird $s : y$ gestrichen. Mit der μ-Operation lassen sich Operationen in Datenräumen, aber auch der Fortschritt der Programmausführung beschreiben: ein Prozeduraufruf ersetzt den Prozedurnamen durch seinen Rumpf; das Prozedurende macht das rückgängig. Das Verfahren ist beschrieben in (Lucas und Walk, 1971) und (Wegner, 1972).

Auf der Basis von VDL entwickelte sich dann die *Vienna Development Method* (VDM), (Björner und Jones, 1978).

4 Die 70er Jahre

Zu Anfang der 70er Jahre war aus dem deutschsprachigen Raum vor allem die ETH Zürich international sichtbar. Niklaus wirth, der 1968 aus Berkeley nach Zürich gekommen war, definierte die Programmiersprache Pascal, (Wirth, 1971), nachdem er zuvor schon die Sprachen Euler, Algol W und die maschinennahe Sprache PL360 geschaffen hatte. Pascal erweiterte Algol um die *record*-Datentypen, *files*, sowie die *case*-Anweisung, die in einfacherer Form schon in Cobol vorhanden waren. Pascal wurde in Zürich implementiert, wobei Wirths Mitarbeiter als Zwischensprache den P-Code definierten, (Nori et al., 1976). Der P-Code ist die Urform der virtuellen Maschinensprachen wie der JVM oder .NET. Ein Interpretierer für den P-Code umfaßte nur wenige 100 Zeilen Maschinensprache; da auch der Übersetzer selbst in P-Code vorlag, konnte man also mit geringem Programmieraufwand ein zwar langsames, aber einwandfrei funktionierendes Pascal-System herstellen. Dies sorgte für die rasche, weltweite Verbreitung der Sprache.

Die Arbeit (Hoare und Wirth, 1973) definierte den Kern der Semantik von Pascal mit Hilfe der Hoare-Logik, zwar ebenfalls operativ, aber anders als VDL. Diese Methodik wurde von vielen weiteren Autoren aufgegriffen, vgl. z.B. (Loeckx et al., 1986).

Die Übersetzerbauer brachten derweil die Arbeiten zum Thema Syntaxanalyse zum Abschluß, z.B. in Form des PGS-Systems von Peter Dencker, einer Karlsruher Diplomarbeit aus dem Jahre 1977, (Dencker et al., 1984), das als Teil des Eli-Systems, (Gray et al., 1992), heute noch in Gebrauch ist.

D. Knuth führte mit der Arbeit (Knuth, 1968, 1971) attributierte Grammatiken (AG) als allgemeine Methodik zur Beschreibung der statischen Semantik von kontextfreien Sprachen ein. Bei Verwendung von rekursivem Abstieg für die Syntaxanalyse erweisen sich

AGs als abstrakte Beschreibungstechnik für bereits seit langem eingesetzte Verfahren für die semantische Analyse.

C.H.A. „Kees“ Koster, einer der Koautoren von Algol 68 und 1972 - 76 Professor an der TU Berlin (später an der Universität Nijmwegen), entwickelte in den frühen 70er Jahren die sogenannten *Affix-Grammatiken*, eine Spezialisierung der van Wijngaarden-Grammatiken, bei der Teile der Nichtterminale von kontextfreien Produktionen als Prozedurparameter gewertet wurden, die bei der Berechnung, z.B. mit rekursivem Abstieg, mit Werten versehen werden konnten. Affix-Grammartiken und das damit konstruierte System CDL zur Konstruktion von Übersetzern war in Deutschland, Belgien, den Niederlanden und Ungarn verbreitet im Einsatz.

Die Erkenntnis, in dieser Notation auch sehr systematisch programmieren zu können, führte zur Erweiterung der Sprache zu CDL2, und zur Entwicklung einer auf CDL basierenden Programmierumgebung. Eingeführt wurde mit CDL2 ein statisches Modulkonzept zur Realisierung eines Schichtenmodells von Software-Architekturen und eine regulierte Kommunikationsstruktur sowohl zwischen Schichten und den Komponenten einer Schicht. Für die unterste Ebene, die so genannte Basisschicht wurden vordefinierte Pakete mit Basisoperationen (z.B. Arithmetik) zur Verfügung gestellt.

Herr Koster definierte dann die Programmiersprache ELAN, (Hommel et al., 1979), die mit CDL2 implementiert wurde. ELAN wurde als Sprache für den Programmierunterricht entwickelt. Zwei wesentliche als Sprachkonstrukte umgesetzte Prinzipien der so genannten strukturierten Programmierung waren die schrittweise Verfeinerung und das Geheimnisprinzip beim Zugriff auf Datenstrukturen.

Das Konzept der schrittweise Verfeinerung wurde als eine Erweiterung des ALGOL60-Blockkonzepts umgesetzt. Blöcke wurden benannt und konnten so aufgerufen werden - allerdings ohne Parameter und umsetzbar durch textuelles Kopieren.

Primär zur Umsetzung des Geheimnisprinzips wurde für die Definition abstrakter Datenstrukturen ein statisches Modulkonzept (Pakete) eingeführt, mit dem Datenstrukturen und Zugriffsoperationen gekapselt werden konnten und der Zugriff auf die Datenstruktur nur mit den definierten Zugriffsoperationen möglich war.

Jochen Liedtke implementierte 1978 Elan für den Prozessor Z80 als Diplomarbeit an der Universität Bielefeld, wobei er die Sprache um (rekursive) Prozesse und Datenräume erweiterte.

ELAN war ähnlich Pascal als Programmiersprache für Schulen empfohlen und wurde zu diesem Einsatzzweck auch von der GMD in Birlinghoven unterstützt. In diesem Zusammenhang entwickelte Jochen Liedtke in einer Erweiterung von ELAN ein Betriebssystem Eumel, ein Mehrbenutzersystem mit virtuellem Speicher auf dem Z80. Das System wurde in Schulen und in Einzelbüros, z.B. bei Rechtsanwälten, viele 1000 Male eingesetzt, auch im Ausland, vor allem in Japan. Eumel wurde auf den Intel 8086 übertragen und hieß dann L3.

Bei einer Demonstration von L3 in Tokio 1985 auf einem Siemens PC konnte Siemens Japan in der Eile keinen Transformator für 100V auftreiben und stellte nur einen Trafo für 110V zur Verfügung. Das Resultat war, daß das System wegen zu geringer Spannung immer wieder abstürzte. Nun speicherte Eumel/L3 seine Daten regelmäßig auf dem Sekundärspeicher; hier geschah das alle 5 Minuten. Beim Neustart hatte das System daher nur die Arbeit der letzten 5 Minuten verloren. Während sich die Deutschen über die Abstürze ärgerten, hatten die Japaner den Eindruck, die Abstürze seien Absicht, um die Erhaltung der Speicherinhalte zu demonstrieren.

Das Nachfolgesystem L4, ein μKern-System, erreichte auf der SOSP 1993 Weltruf,

als Liedtke nachwies, daß seine Implementierung des Fernaufrufs, (*remote procedure call*), 20 mal schneller war als die Implementierung im CMU μKern Mach, (Liedtke, 1993). Die amerikanischen Kollegen hielten das für unglaublich, bis es ihnen demonstriert wurde. Gernot Heiser und seine Mitarbeiter an der *University of New South Wales* bewiesen die Korrektheit von L4, (Klein et al., 2010). Es wird heute millionenfach eingesetzt, u.a. in *smartphones*.

5 Die 80er Jahre

Uwe Kastens konnte in seiner Dissertation 1976, (Kastens, 1980), an der Universität Karlsruhe *geordnete attributierte Grammatiken* (OAG) definieren, die sich als ausreichend für die Beschreibung der semantischen Analyse von beliebigen Programmiersprachen erwiesen. Geordnete attributierte Grammatiken sind ein polynomiell berechenbarer Spezialfall von partitionierten attributierten Grammatiken, (Waite und Goos, 1984), einer Eigenschaft, die im allgemeinen nur mit Aufwand NP festgestellt werden kann. Die Eigenschaft garantiert, daß man programmunabhängig die Reihenfolge festlegen kann, in der die Attribute eines abstrakten Syntaxbaums berechnet werden können. Mit dem GAG-System, (Kastens et al., 1982), implementierte Kastens diese Reihenfolgebestimmung für OAGs samt der semantischen Analyse für Anwendungsprogramme. Eine weiter entwickelte Form von GAG, das Liga-System, ist Teil des bereits zitierten Eli-Systems, (Gray et al., 1992).

Damit waren die Karlsruher in der Lage die semantische Analyse von Ada komplett zu spezifizieren und zu testen, bevor sie eine effiziente Implementierung vornahmen, (Uhl et al., 1982). Dies verschaffte den Karlsruhern einen großen zeitlichen Vorsprung bei ihrer Implementierung von Ada 83. Sie steuerten überdies einen großen Teil der inhaltlichen Spezifikation der Zwischensprache Diana bei, einem *de facto* Standard für die interne Darstellung von Ada 83-Programmen, (Goos und Wulf, 1983).

6 Die 90er Jahre und danach

Reinhard Wilhelm und seine Mitarbeiter an der Universität des Saarlandes brachten die statische Programmanalyse in mehreren Punkten weiter: Zusammen mit Mooly Sagiv, Tel Aviv, und Thomas Reps, Madison, entwickelte Herr Wilhelm eine statische Programmanalyse auf der Grundlage einer 3-wertigen Logik, (Sagiv et al., 2002), die sogenannte *shape*-Analyse. Der Ausgangspunkt dieser Entwicklung war der Versuch, herauszufinden, wie Programme verzeigerte Datenstrukturen auf der Halde manipulieren. Eine solche Aussage könnte sein, daß ein Programm dem man eine doppelt verkettete Liste mit einem Kopfzeiger und ohne weitere Zeiger gibt, eine solche Liste zurückgibt. Die besonderen Probleme dabei sind, daß solche Datenstrukturen aus anonymen Objekten bestehen und daß die Belegung der Halde prinzipiell unendlich wachsen kann. Mithilfe der entwickelten kanonischen Abstraktion kann jeder Haldeninhalt beliebiger Größe auf einen abstrakten Inhalt beschränkter Größe abgebildet werden. Es zeigte sich, daß die kanonische Abstraktion noch weit mehr schwierige Programmeigenschaften approximativ heraus finden kann, z.B. Synchronisationseigenschaften nebenläufiger Programme.

Ab Ende der 90er Jahre entwickelte die Gruppe von Reinhard Wilhelm die erste Lösung des Problems, die Echtzeiteigenschaften von eingebetteten Programmen und komplexen

Prozessorarchitekturen nachzuweisen, (Ferdinand et al., 2001). Wesentliche Bestandteile der Technologie sind mehrere statische Programmanalysen, also Techniken aus dem Übersetzerbau. Vollkommen neu sind die statischen Analysen, welche an allen Programmpunkten Invarianten über die Menge der dort möglichen Ausführungszustände berechnen, z.B. die Menge der dort möglichen Cacheinhalte. Mithilfe solcher Invarianten lassen sich die bei modernen Prozessoren extrem hohen Schwankungen der Ausführungszeiten einschränken. Die Firmenausgründung AbsInt industrialisierte diese Entwicklung und bietet bis heute die einzigen industriell weithin eingesetzten Werkzeuge zur Echtzeitverifikation an.

7 Danksagung

Die Herren Stefan Jähnichen, Hans Langmaack, Jacques Loeckx und Reinhard Wilhelm haben durch ihre Beiträge und Kritik erheblich zu diesem Papier beigetragen.

Literatur

Bauer, F. (1960): The Formula-controlled Logical Computer "Stanislaus". *Math. Comput.*, 14: 64–67.

Bauer, F. L. (1977): Angstl's Mechanism for Checking Wellformedness of Parenthesis-Free Formulae. *Math. Comp.*, 31(137): 318–320.

Bauer, F. L. (1994): Die ALGOL-Verschwörung. Techn. Ber. 9409, CAU Kiel.

Bauer, F. L. und Samelson, K. (1957): *Patentschrift: Verfahren zur automatisierten Verarbeitung von kodierten Daten und Rechenmaschine zur Ausübung des Verfahrens*. Deutsches Patentamt. Patent 1 094 019.

Bauer, F. L. und Wössner, H. (1972): Zuses „Plankalkül", ein Vorläufer der Programmiersprachen — gesehen vom Jahre 1972. *Elektronische Rechenanlagen*, 14(3): 111–118.

Baumann, R. (1961): ALGOL-*Manual der* ALCOR-*Gruppe*. Oldenbourgh-Verlag, München.

Bayer, R., Gries, D., Paul, M. und Wiehle, H. R. (1967): The ALCOR Illinois 7090/7094 Post Mortem Dump. *Commun. ACM*, 10(12): 804–808.

Björner, D. und Jones, C. B. (1978): *The Vienna Development Method: The Meta-Language*, Bd. 61 von *Lecture Notes in Computer Science*. Springer.

Dahl, O. J., Myhrhaug, B. und Nygaard, K. (1968): *Simula 67, Common Base Language*. Norwegisches Rechenzentrum, Oslo.

Dencker, P., Dürre, K. und Heuft, J. (1984): Optimization of Parser Tables for Portable Compilers. *ACM Transactions on Programming Languages and Systems*, 6(4): 546–572.

Eickel, J., Paul, M., Bauer, F. L. und Samelson, K. (1963): A syntax controlled generator of formal language processors. *Commun. ACM*, 6(8): 451–455.

Ferdinand, C., Heckmann, R., Langenbach, M. et al. (2001): Reliable and Precise WCET Determination for a Real-Life Processor. In *Embedded Software, First International Workshop, EMSOFT 2001, Tahoe City, CA, USA, October, 8-10, 2001, Proceedings*, herausgegeben von Henzinger, T. A. und Kirsch, C. M., Bd. 2211 von *Lecture Notes in Computer Science*, S. 469–485. Springer. ISBN 3-540-42673-6.

Gödel, K. (1931): Über Formal Unentscheidbare Sätze Der Principia Mathematica Und Verwandter Systeme I. *Mh. Math. Phys.*, 38: 173–198.

Goos, G. und Wulf, W. A. (Hrsg.) (1983): *Diana, an Intermediate Language for Ada*, Bd. 161 von *LNCS*. Springer.

Grau, A., Hill, U. und Langmaack, H. (1967): *Translation of Algol 60*, Bd. 1b von *Handbook for automatic computation*. Springer, Heidelberg.

Gray, R. W., Heuring, V. P., Levi, S. P. et al. (1992): Eli: A Complete, Flexible Compiler Construction System. *Comm. ACM*, 35(2): 121–131. Eli is available at http://www.cs.colorado.edu/ēliuser/ or http://www.uni-paderborn.de/project-hp/eli.html.

Hill, U., Langmaack, H., Schwarz, H. und Seegmüller, G. (1962): Efficient handling of subscripted variables in ALGOL 60 compilers. In *Proc. 1962 Rome Symposium on Symbolic Languages in Data Processing*, S. 311–340, New York. Gordon and Breach.

Hoare, C. (1973): Hints on Programming Language Design. In *Symposium on Principles of Programming Languages*. ACM. reprinted in (Horowitz, 1983, pp. 31 – 40).

Hoare, C. A. R. und Wirth, N. (1973): An Axiomatic Definition of the Programming Language Pascal. *Acta Informatica*, 2: 335–355.

Hommel, G., Jäckel, J., Jähnichen, S. et al. (1979): *ELAN-Sprachbeschreibung*. Akademische Verlagsgesellschaft, Wiesbaden.

Horowitz, E. (Hrsg.) (1983): *Programming Languages — A Grand Tour*. Springer.

IEEE 754 (2008): IEEE Standard for binary floating-point arithmetic. Techn. Ber., ANSI/IEEE. Std. 754 - 2008.

Kastens, U. (1980): Ordered Attribute Grammars. *Acta Informatica*, 13(3): 229–256.

Kastens, U., Hutt, B. und Zimmermann, E. (1982): *GAG: A Practical Compiler Generator*. Nr. 141 in Lecture Notes in Computer Science. Springer Verlag.

Klein, G., Andronick, J., Elphinstone, K. et al. (2010): seL4: Formal Verification of an Operating-System Kernel. *Commun. ACM*, 53(6): 107–115.

Knuth, D. E. (1968): Semantics of Context-Free Languages. *Mathematical Systems Theory*, 2(2): 127–146. The invention of AGs.

Knuth, D. E. (1971): Semantics of Context-Free Languages: Correction. *Mathematical Systems Theory*, 5: 95–96. The invention of AGs.

Liedtke, J. (1993): Improving IPC by Kernel Design. In *14th ACM Synposium on Operating System Principles*, Asheville, North Carolina. ACM.

Loeckx, J., Mehlhorn, K. und Wilhelm, R. (1986): *Grundlagen der Programmiersprachen*. Teubner, Stuttgart.

Lucas, P. und Walk, K. (1971): On the Formal Description of PL/I. *Annual Review of Automatic Programming*, 6: 105–182.

McCarthy, J. (1960): Recursive Functions of Symbolic Expressions and their Computation byMachine. *Communications of the ACM*, 3(4): 184 – 195.

Naur P. (Hrsg.) (1960): Report on the Algorithmic Language Algol 60. *Numer. Math.*, 2: 106–136. und Comm. ACM 3(1960), 299-314.

Naur P. (Hrsg.) (1962): Revised Report on the Algorithmic Language Algol 60. *Numer. Math.*, 4: 420–453. und Comm. ACM 6(1963), 1-17.

Nori, K. V., Ammann, U., Jensen, K. et al. (1976): The PASCAL <P> Compiler: Implementation Notes. Techn. Ber., Institut für Informatik, ETH Zürich.

Paul, M. (1962): *Zur Struktur formaler Sprachen*. Dissertation, Universität Mainz.

Perlis, A. J. und Samelson, K. (Hrsg.) (1958): Preliminary report: International algebraic Language. *CACM*, 1(12): 8–22.

Perlis, A. J. und Samelson, K. (Hrsg.) (1959): Report on the Algorithmic Language ALGOL. *Numerische Mathematik*, 1: 41–60.

Rutishauser, H. (1951): Über automatische Rechenplanfertigung bei programmgesteuerten Rechenmaschinen. *Z. angew. Math. Mech.*, 31: 255ff.

Sagiv, M., Reps, T. und Wilhelm, R. (2002): Parametric Shape Analysis via 3-Valued Logic. *ACM Transactions on Programming Languages and Systems*, 24(3): 217–298.

Samelson, K. (1957): Probleme der Programmierungstechnik. In *Internationales Kolloquium über Probleme der Rechentechnik 1955*, S. 61–68, Berlin.

Samelson, K. und Bauer, F. L. (1959): Sequentielle Formelübersetzung. *Elektron. Rechenanlagen*, 1: 176–182.

Samelson, K. und Bauer, F. L. (1960): Sequential Formula Translation. *Commun. ACM*, 3(2): 76–83.

Seegmüller, G. (1962): Some Remarks on the Computer as a Source Language Machine. In *Proc. IFIP Congress*, Bd. 62, S. 524–525.

Uhl, J., Drossopoulou, S., Persch, G. et al. (1982): *An Attribute Grammar for the Semantic Analysis of Ada*, Bd. 139 von *LNCS*. Springer.

Waite, W. M. und Goos, G. (1984): *Compiler Construction*. Springer Verlag.

Wegner, P. (1972): The Vienna Definition Language. *Comp. Surveys*, 4(1): 5–63.

Wilkes, M., Wheeler, D. J. und Gill, S. (1951): *The Preparation of Programs for an Electronic Digital Computer*. Addison-Wesley.

Wirth, N. (1971): The Programming Language Pascal. *Acta Informatica*, 1: 35–63.

Zuse, K. (1972): Der Plankalkül. Bericht Nr. 63, GMD, St. Augustin.

DeLICM – Undoing SSA Transformations for Polly

Michael Kruse

INRIA, France
michael.kruse@inria.fr

Abstract. The LLVM compiler and its intermediate representation are SSA-based. Most of its optimizations assume virtual registers and therefore the framework tries to convert as many load/store chains to virtual registers as possible. Unfortunately, this adds obstacles for polyhedral optimizers such as Polly. Virtual registers are 0-dimensional and as such often sequentialize otherwise parallel code. This is especially true for scalar optimizations that move code around such as Loop-Invariant Code Motion and Partial Reduction/Common Subexpression Elimination. Polly's current approach is to analyze the code before these scalar optimizations occur. We show that it is possible to undo the effects of these passes on the polyhedral level, which executing Polly later in LLVM pipeline where more SCoPs can be detected, without the additional scalar dependencies.

Keywords: Static Single Assignment, Polyhedral Compilation, Loop-Invariant Code Motion, Partial Redundancy Elimnation, LLVM, Polly

Security Through Safety

Benedikt Nordhoff

Westfälische Wilhelms-Universität
Münster, Germany

In the first section, we will define the program model and security property we are aiming at. The second section contains a characterization of critical executions which must exist for the security property to be violated and has been formalized in the interactive theorem prover Isabelle/HOL. This is then used in the last section to construct a safety analysis that asserts the security property.

1 Security Property

For the definition of our program model let Var be a finite set of variables and Val be an arbitrary set of values. The states on which our programs operate are maps from variables to values $\Sigma = \text{Var} \to \text{Val}$. We assume the confidential input is part of the initial state as values of a set $H \subseteq \text{Var}$ of initially *high* variables. We call the remainder $L = \text{Var} \setminus H$ initially *low* variables. For an arbitrary set $X \subseteq \text{Var}$ of variables and two states $\sigma, \sigma' \in \Sigma$ we denote by $\sigma =_X \sigma'$ the pointwise equality of σ and σ' for all $x \in X$ and in the special case that $X = L$ we call the states σ and σ' *low equal*. We use control flow graphs for our program model.

Definition 1 (Program model). *A control flow graph is a tuple $(N, E, \mathfrak{e}, \mathfrak{r}, [\![.]\!])$ where N is a finite set program locations, $[\![.]\!] : N \times \Sigma \to N \times \Sigma$ is the deterministic semantics function, $E \subseteq N \times N$, is a safe control flow abstraction i.e. $\forall n, n'.\ \exists \sigma, \sigma'.\ [\![n, \sigma]\!] = (n', \sigma') \Rightarrow (n, n') \in E$. $\mathfrak{e}, \mathfrak{r} \in N$ are the initial respectively final location and we require, that $N \times \{\mathfrak{r}\} \subseteq E^*$, $\forall n.\ (\mathfrak{r}, n) \in E \Rightarrow n = \mathfrak{r}$ and $\forall \sigma. [\![\mathfrak{r}, \sigma]\!] = (\mathfrak{r}, \sigma)$.*

For a location state tuple $x \in N \times \Sigma$, we denote by x_l (respectively x_s) the projection to the program location (respectively the state). We assume that we have two auxiliary functions $\text{use} : N \to 2^{\text{Var}}$, which models the sets of variables used, and $\text{def} : N \to 2^{\text{Var}}$, which model the sets of variables defined by an instruction in the canonical way. That is $\text{def}(n) \supseteq \{v \in \text{Var} \mid \exists \sigma.\ \sigma(v) \neq [\![n, \sigma]\!]_s(v)\}$ and $\text{use}(n) \subseteq \text{Var}$, such that for all σ, σ' that satisfy $\sigma =_{\text{use}(n)} \sigma'$ it holds $[\![n, \sigma]\!]_l = [\![n, \sigma']\!]_l$ and $[\![n, \sigma]\!]_s =_{\text{def}(n)} [\![n, \sigma']\!]_s$.

Definition 2 (Execution). *A formal execution is a path $\pi = (\pi_i)_{0 \leq i} \in N^\omega$ where $\pi_0 = \mathfrak{e}$ and for all $0 < i$ it holds that $(\pi_{i-1}, \pi_i) \in E$. It is called feasible from some state σ_0 if $\forall i.\ \pi_i = [\![\pi_0, \sigma_0]\!]_l^i$ and in this case $([\![\pi_0, \sigma_0]\!]_s^i)_{0 \leq i}$ is called the corresponding state sequence. An execution is called terminating if there exists an index k such that $\pi_k = \mathfrak{r}$.*

Definition 3 (Attacker). *An attacker is modeled as a uniquely defined relation (partial function) of observables* $\text{obs} \subseteq N \times (\Sigma \to \text{Val})$ *such that for all* $(n, f) \in \text{obs}$ *the value of* f *only depends upon the values of the variables in* $\text{use}(n)$*, that is:* $\forall \sigma, \sigma' \in \Sigma.\ \sigma =_{\text{use}(n)} \sigma' \Rightarrow f(\sigma) = f(\sigma')$*. The observation made by the attacker from an initial state* σ_0 *is defined as follows. Let* $\pi = \pi_0 \pi_1 \ldots$ *be the execution corresponding to* σ_0*,* $(\sigma_i)_{1 \leq i}$ *be the corresponding states and* $(i_j)_{1 \leq j < l}$ *for* $l \in \mathbb{N} \cup \{\infty\}$ *be the maximal increasing family of indices such that* $\pi_{i_j} \in \text{dom}(\text{obs})$ *for all* j*. Then the sequence* $\text{obs}(\sigma_0) := (\text{obs}(\pi_{i_j})(\sigma_{i_j}))_{1 \leq j < l}$ *is the observation made from* σ_0 *by the attacker* obs*.*

Definition 4 (Security). *A program is called* α^L_{obs}*-secure if for all pairs of low equivalent states* $\sigma =_L \sigma'$*, for which the corresponding maximal executions terminate, it holds that* $\text{obs}(\sigma) = \text{obs}(\sigma')$*.*

A program is called β^L_{obs}*-secure if for all pairs of low equivalent states* $\sigma =_L \sigma'$*, it holds that* $\text{obs}(\sigma) \leq \text{obs}(\sigma')$ *or* $\text{obs}(\sigma') \leq \text{obs}(\sigma)$*. Here* $\leq$ *is the prefix order on sequences.*

Note that $\alpha + \beta$ security corresponds to indistinguishable security [2]. We don't require that non-terminating executions produce the same observation, as this would require the analysis to prove termination of all loops influenced by high data.

2 Characterization

Let Π denote the set of infinite paths in E. That is $\Pi = \{(\pi_i)_{0 \leq i} \mid \forall 0 < i.\ (\pi_{i-1}, \pi_i) \in E\}$. Let $\xrightarrow{pd} \subseteq N \times N$ be the post dominance relation, that is $n \xrightarrow{pd} m$ iff $\forall (\pi_i)_{0 \leq i} \in \Pi, k.\ \pi_0 = m \wedge \pi_k = \mathfrak{r} \Rightarrow \exists i \leq k.\ \pi_i = n$. As we assumed, that $\mathfrak{r}$ can be reached from all locations and doesn't reach any other locations, the post dominance relation $\xrightarrow{pd}$ is the transitive reflexive closure of the post dominance tree rooted in $\mathfrak{r}$ and the immediate post dominator $\text{ipd}(m)$ is well defined for all $m \in N \setminus \{\mathfrak{r}\}$ as the unique location $n \neq m$ such that $n \xrightarrow{pd} m$ and $\forall n'.\ n' \neq m \wedge n' \xrightarrow{pd} m \Rightarrow n' \xrightarrow{pd} n$.

We define control dependencies in the style of Xin & Zhang [3] based on regions which correspond to a path from a branching point up to the corresponding immediate post dominator and use the standard definition of data dependency. For a path $\pi \in \Pi$ the control dependency relation $\xrightarrow{cd_\pi} \subseteq \mathbb{N} \times \mathbb{N}$ is defined as follows $i \xrightarrow{cd_\pi} k$ iff $i < k \wedge \pi_k \neq \mathfrak{r} \wedge \forall j.\ i \leq j \leq k \Rightarrow \pi_j \neq ipd(\pi_i)$. With this we can define the control slice of index k in $\pi \in \Pi$ as $cs^\pi_k = (\pi_{i_j})_{0 \leq j \leq l}$ where $(i_j)_{0 \leq j \leq l}$ is the maximal increasing sequence of indices such that $i_l = k$ and for all $0 \leq j < j' \leq l$ $i_j \xrightarrow{cd_\pi} i_{j'}$. The control slice is useful to identify occurrences of the same location in different executions as it has the useful property: $\forall i, i', j, j' \in \mathbb{N}, \pi, \pi' \in \Pi.\ \text{cs}^\pi_i = \text{cs}^{\pi'}_{i'} \neq \mathfrak{r} \wedge \text{cs}^\pi_j = \text{cs}^{\pi'}_{j'} \wedge i < j \Rightarrow i' < j'$.

For a path $\pi \in \Pi$ and variable $v \in \text{Var}$ the data dependency relation $\xrightarrow{dd_{\pi,v}} \subseteq \mathbb{N} \times \mathbb{N}$ is defined by $i \xrightarrow{dd_{\pi,v}} k$ iff $i < k \wedge v \in \text{def}(\pi_i) \cap \text{use}(\pi_k) \wedge \forall j \in [i+1, k-1].\ v \notin \text{def}(\pi_j)$.

We use the following characterization to identify critical executions. For a state σ let $\pi^\sigma = (\pi_i^\sigma)_{0\leq i}$ denote the corresponding path with state sequence $(\sigma_i)_{0\leq i}$. We define the the set of conflict pairs $cp \subseteq (\Sigma \times \mathbb{N})^2$ as the smallest relation closed under rules defined in equations (1) – (4).

$$\frac{\begin{array}{c}\sigma =_L \sigma' \wedge \mathrm{cs}_i^{\pi^\sigma} = \mathrm{cs}_{i'}^{\pi^{\sigma'}} \wedge \\ h \in \mathrm{use}(\pi_i^\sigma) \wedge \sigma_i(h) \neq \sigma'_{i'}(h) \wedge \\ \forall j < i.\ h \notin \mathrm{def}(\pi_j^\sigma) \wedge \\ \forall j' < i'.\ h \notin \mathrm{def}(\pi_{j'}^{\sigma'})\end{array}}{(\sigma, i)\ cp\ (\sigma', i')} \tag{1}$$

$$\frac{\begin{array}{c}(\sigma, j)\ cp\ (\sigma', j') \wedge \mathrm{cs}_i^{\pi^\sigma} = \mathrm{cs}_{i'}^{\pi^{\sigma'}} \wedge \\ j \xrightarrow{dd_{\pi,v}} i \wedge j' \xrightarrow{dd_{\pi',v}} i' \wedge \\ \sigma_i(v) \neq \sigma'_{i'}(v)\end{array}}{(\sigma, i)\ cp\ (\sigma', i')} \tag{2}$$

$$\frac{\begin{array}{c}(\sigma, j)\ cp\ (\sigma', j') \wedge \mathrm{cs}_i^{\pi^\sigma} = \mathrm{cs}_{i'}^{\pi^{\sigma'}} \wedge \\ j \xrightarrow{cd_\pi} k \wedge k \xrightarrow{dd_{\pi,v}} i \wedge \\ \pi^\sigma_{j+1} \neq \pi^{\sigma'}_{j'+1} \wedge \sigma_i(v) \neq \sigma'_{i'}(v) \wedge \\ \forall l' \in [\inf\{\iota' | j' < \iota' \wedge \exists \iota.\ \mathrm{cs}_\iota^{\pi^\sigma} = \mathrm{cs}_{\iota'}^{\pi^{\sigma'}}\}, i' - 1].\ v \notin \mathrm{def}(\pi_{l'}^{\sigma'})\end{array}}{(\sigma, i)\ cp\ (\sigma', i')} \tag{3}$$

$$\frac{(\sigma', i')\ cp\ (\sigma, i)}{(\sigma, i)\ cp\ (\sigma', i')} \tag{4}$$

We obtain the property, that, if the executions for two low equal states reach the same control slice but read different values for some variable, then they create a conflicting pair.

Lemma 1. $\sigma =_L \sigma' \wedge \mathrm{cs}_i^{\pi^\sigma} = \mathrm{cs}_{i'}^{\pi^{\sigma'}} \wedge \sigma_i \neq_{\mathrm{use}(\pi_i^\sigma)} \sigma'_{i'} \Rightarrow (\sigma, i)\ cp\ (\sigma', i')$

Based on this we define observable conflict pairs as those, that either both reached an observable node or where one has but the other can not reach an observable node. We define the the set of observable conflict pairs $cop \subseteq (\Sigma \times \mathbb{N})^2$ as the smallest set closed under the following rules:

$$\frac{(\sigma, i)\ cp\ (\sigma', i') \wedge \pi_i^\sigma \in \mathrm{dom}(\mathrm{obs})}{(\sigma, i)\ cop\ (\sigma', i')} \tag{5}$$

$$\frac{(\sigma, j)\ cp\ (\sigma', j') \wedge j \xrightarrow{cd_\pi} i \wedge \pi^\sigma_{j+1} \neq \pi^{\sigma'}_{j'+1} \wedge \pi_i^\sigma \in \mathrm{dom}(\mathrm{obs})}{(\sigma, i)\ cop\ (\sigma', j')} \tag{6}$$

Theorem 1. *If there do not exist any states σ, σ' and indices i, i' such that $\sigma =_L \sigma'$ and $(\sigma, i)\ cop\ (\sigma', i')$ then the program is α^L_{obs} and β^L_{obs} secure.*

3 Analysis

Inspecting equations (1) – (6) we can obtain a sound approximation for individual potentially critical executions. Equation (1) is handled by the initialization $I_\pi = \{i \in \mathbb{N} | \exists h \in H \cap \text{use}(\pi_i).\ \forall j < i.\ h \notin \text{def}(\pi_j)\}$. To handle the executions from equation (2) we can use the previously defined data dependencies. In equation (3) the execution π^σ is captured by control dependencies plus data dependencies and to capture execution $\pi^{\sigma'}$ we use data control dependencies defined in equation (7) below.

$$i \xrightarrow{dcd_{\pi,v}} k \Leftrightarrow \exists \pi' i' j' k'.\ \text{cs}_i^\pi = \text{cs}_{i'}^{\pi'} \wedge \text{cs}_k^\pi = \text{cs}_{k'}^{\pi'} \wedge i' \xrightarrow{cd_{\pi'}} j' \xrightarrow{dd_{\pi',v}} k' \quad (7)$$

Theorem 2. *If there exists no feasible execution π and index i such that $\pi_i \in \text{dom(obs)}$ and $I_\pi \times \{i\} \cap (\bigcup\{\xrightarrow{cd_\pi}, \xrightarrow{dd_{\pi,v}}, \xrightarrow{dcd_{\pi,v}} | v \in \text{Var}\})^* \neq \emptyset$ then the program is α^L_{obs} and β^L_{obs} secure.*

Our prototypical implementation within the CPAchecker framework [1] corresponds to a finite automaton that uses five types of states to track control, data and data control dependencies within an execution and marks potentially critical states as target states who's reachability is then tried to be refuted by different analyses in the CPAchecker framework.

The analysis starts with the special state **Init**. The state **Target** marks a potentially critical state. States of the form $\mathbf{DD(v)}$ are used to track a data dependency on the variable v, the state $\mathbf{CD(m)}$ tracks a control dependency that stretches up to node m which is the immediate post dominator of some branching node n that introduced this control dependency, and the state $\mathbf{DCD(m, v)}$ denotes a data control dependency where m is the immediate post dominator of the branching node that introduced this data control dependency and v is the variable that could have be written on an alternative branch as identified by a static preanalysis.

The transfer relation on these states is defined by $s \xrightarrow{n} t \Leftrightarrow Crit(s, n, t) \wedge Intro(t, n, s) \vee s = t \wedge P(s, n)$ where $Crit$, $Intro$ and P are defined as in Table 1.

Table 1. Rules for the transfer relation of the prototypical implementation.

s	**Intro(s, n, f)**	**Crit(s, n, t)**	**P(s, n)**
Init	$false$	$H \cap \text{use}(n) \neq \emptyset$	$true$
DD(v)	$v \in \text{def}(n) \vee isDCD(v, f)$	$v \in \text{use}(n)$	$v \notin \text{def}(n)$
CD(m)	$m = \text{ipd}(n)$	$m \neq n$	$m \neq n$
DCD(m, v)	$m = \text{ipd}(n) \wedge \exists \xrightarrow{cd_v}$ before m	$m = n \wedge isDD(v, t)$	$m \neq n$
Target	n is observable	$false$	$false$

References

1. Beyer, D., Keremoglu, M.E.: Cpachecker: A tool for configurable software verification. In: Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. pp. 184–190 (2011), https://doi.org/10.1007/978-3-642-22110-1_16
2. Bohannon, A., Pierce, B.C., Sjöberg, V., Weirich, S., Zdancewic, S.: Reactive noninterference. In: Proceedings of the 16th ACM Conference on Computer and Communications Security. pp. 79–90. CCS '09, ACM, New York, NY, USA (2009), http://doi.acm.org/10.1145/1653662.1653673
3. Xin, B., Zhang, X.: Efficient online detection of dynamic control dependence. In: Proceedings of the 2007 international symposium on Software testing and analysis. pp. 185–195. ISSTA '07, ACM, New York, NY, USA (2007), http://doi.acm.org/10.1145/1273463.1273489

Modelling Haskell Programs with Free Monads (Extended Abstract)

Sandra Dylus[1] and Jan Christiansen[2]

[1] University of Kiel
sad@informatik.uni-kiel.de
[2] University of Applied Science Flensburg
jan.christiansen@hs-flensburg.de

When we want to prove statements about a Haskell program in an interactive theorem prover like Agda or Coq, we have to transform the Haskell program into an Agda/Coq program. As Agda/Coq programs have to be total and Haskell programs are often not, we have to model partiality explicitly in the target language. In a preceding work Abel et al. (2005) propose such a translation from Haskell to Adga, which totalises potentially partial Haskell programs. A natural way of modelling this partiality is to use the `Maybe` monad. A more general approach is to generalise the monad at play. Indeed, the translation of Abel et al. (2005) uses a monadic lifting of Haskell programs to define partial as well as total Haskell functions in Adga. That is, functions are applied to monadic arguments and yield a monadic result. Since functions yield monadic values, the next step is to lift data type declarations as well. That is, each argument of a data type constructor becomes a monadic value. Monadic values, which are results of the lifted functions, can then be used directly in these lifted data types. All in all, the transformation produces monad generic definitions for total functions; partial functions cannot be defined for any monad, thus, partial functions are explicitly instantiated by `Maybe`.

For the actual proof of properties, the monadic context is then explicitly instantiated with the `Identity` monad for corresponding total Haskell definitions. If any partial functions are involved, we have to instantiate all functions at play with `Maybe`. Therefore, properties have to be explicitly proven in the appropriate context. Properties about functions instantiated with `Identity` cannot be reused when arguing about partial functions instantiated with `Maybe` and vice versa. Moreover, in the work of Abel et al. (2005) all properties of interest are proven for both instantiations, `Maybe` and `Identity`, respectively. As a consequence, even properties that hold independent of the underlying monadic effect are proven for both concrete instantiations.

A first try to implement a recursive Haskell data type – for example lists – in Coq following their approach failed. A recursive data type that is parametrised over a generic monad M fails to compile in Coq because of a *non strictly positive occurrence*. This strict positivity restriction for inductive data type declarations is also implemented in more recent versions of Agda, that is, this problem is not a specific to Coq, but a general restriction regarding state-of-the-art dependently typed languages. The underlying logic of the dependently typed language would

become inconsistent if we allow inductive data types that do not comply to strict positivity.

In this work we propose to reanimate the approach of Abel et al. (2005) by using known solutions concerning strict positivity and show how to prove properties of Haskell programs in Coq. Besides the reanimation of the original approach, we argue that due to our modelling, we are now also able to prove monad generic properties without an unreasonable extra effort.

The first idea is to use free monads as introduced by Swierstra (2008) to represent the possible monadic effect instead of a generic monadic parameter M. Free monads capture the essence of a monad, that is, the operations `>>=` and `return`, in a data type that is parametrised over a functor F. This functor has to be strictly positive in order to define the free monad as an inductive data type. Hence, we define the free monad using containers as presented by Abbott et al. (2003). The same idea was realised by Keuchel and Schrijvers (2013) to define fix points of data types for generic programming in Coq.

In order to demonstrate the applicability of our approach we show several exemplary Haskell functions and properties for these functions. These examples demonstrate the benefit of monad-generic proofs: although some of the considered functions are partial, we prove helping lemmas for all monads. This way we can reuse these lemmas when considering statements in a total as well as in a partial setting. However, we will observe that some statements about Haskell functions only hold for all possible effects if the function at hand satisfies additional requirements. These requirements are well-known from other effectful programming languages like functional logic and probabilistic programming languages.

References

M. Abbott, T. Altenkirch, and N. Ghani. *Categories of Containers*, pages 23–38. Springer Berlin Heidelberg, 2003.

A. Abel, M. Benke, A. Bove, J. Hughes, and U. Norell. Verifying haskell programs using constructive type theory. In *Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 62–73, 2005.

S. Keuchel and T. Schrijvers. Generic datatypes à la carte. In *Proceedings of the 9th ACM SIGPLAN Workshop on Generic Programming*, pages 13–24, 2013.

W. Swierstra. Data types à la carte. *Journal of functional programming*, 18(04): pages 423–436, 2008.

The Intended Meaning of *Undefined Behaviour* in C Programs

M. Anton Ertl*

TU Wien

Abstract. All significant C programs contain undefined behaviour. There are conflicting positions about how to deal with that: One position is that all these programs are broken and may be compiled to arbitrary code. Another position is that tested and working programs should continue to work as intended by the programmer with future versions of the same C compiler. In that context the advocates of the first position often claim that they do not know the intended meaning of a program with undefined behaviour. This paper explores this topic in greater depth. The goal is to preserve the behaviour of existing, tested programs. It is achieved by letting the compiler define a consistent mapping of C operations to machine code; and the compiler then has to stick to this behaviour during optimizations and in future releases.

1 Introduction

The C standard does not define the behaviour of most C programs; e.g., it does not define the behaviour of any terminating C program, nor of any library function not defined in the C standard. Some of these behaviours that the C standard does not define are called *undefined behaviour*. And while the C standard makes no difference between these and other non-standard behaviours as far as program conformance is concerned[1], language lawyers and some compiler maintainers are especially fond of *undefined behaviour*.

There are 203 undefined behaviours listed in appendix J of the C11 standard, and most C programs may perform some undefined behaviour under some circumstances, some of them unintended (e.g., buffer overflow vulnerabilities), some of them intended (e.g., a check protecting against a buffer overflow that contains an address computation overflow intended to wrap around).

* Correspondence Address: Institut für Computersprachen, Technische Universität Wien, Argentinierstraße 8, A-1040 Wien, Austria; anton@mips.complang.tuwien.ac.at

[1] Any working C program is a *conformant C program*, and no program with any kind of behaviour not defined in the C standard (e.g., no terminating program) is a *strictly conformant C program*.

A huge number of C programs[2] performs undefined behaviours, even some[3] maintained by the very people who advocate[4] that no C program should perform undefined behaviour.

In the good old days compilers generated code consistently. The behaviour of the generated code for the undefined-behaviour cases was a straightforward extension of the behaviour for the normal case (see Section 4).

Unfortunately, over the last two decades a significant number of compiler maintainers have taken the attitude that their compilers can assume that the programs they compile don't perform undefined behaviour, and this leads to undesirable consequences in combination with sophisticated optimization.[5] In the words of John Regehr[6]: "A sufficiently advanced compiler is indistinguishable from an adversary." In the rest of this paper, such compilers are called *adversarial compilers*.

What to do about it? Some people suggest "fixing" all programs with undefined behaviour, and propose to find them using "sanitizers", i.e., run-time checkers that report some of the undefined behaviours that occur during a given execution run.

An alternative approach is to actually compile these programs as intended. Then programmers can spend their valuable time on hunting down and fixing the remaining bugs, on increasing the performance effectively and/or on extending the functionality of the program. One claim that the advocates of adversarial compilers have made is that they don't know the intended meaning of programs with undefined behaviour, going as far as claiming that it would take psychic powers[7] to compile the code as intended by the programmers.

In this paper I take a closer look at what the intended meaning of C programs is. Section 2 looks at previous work. Section 3 explains that the dream of a totally defined C is not realistic because of architectural differences. Section 4 gives a basic, tight model of the intended meaning. Section 5 discusses whether, how and why that model can be loosened while still staying within the intended meaning of existing, tested programs, and how that affects optimizations. Section 6 looks at how this applies to C library functions, using `memcpy()` as example.

[2] Dan Bernstein estimates 100% https://groups.google.com/forum/message/raw?msg=boring-crypto/48qa1kWignU/b7WsAezEAwAJ.
[3] http://blog.regehr.org/archives/761
[4] http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html
[5] E.g., OpenSSH had a `memset()` intended to erase the private key, i.e., to mitigate the effects of some vulnerability elsewhere; the compiler "optimized" this `memset()` away, because it would require undefined behaviour to read this private key; unfortunately, OpenSSH contained such a vulnerability, the undefined behaviour happened, and private keys were compromised. https://lwn.net/Articles/672465/
[6] https://blog.regehr.org/archives/970
[7] <news:hO2dnSk5W8n4p57OnZ2dnUVZ_qSdnZ2d@supernews.com>

2 Previous work

Wang et al. [WCC+12] give a number of examples of the problems that adversarial compilers cause to existing programs, and provide performance numbers for the effect of optimizations based on adversarial assumptions.

Some people have reacted to adversarial compilers by trying to initiate a tighter definition of C.

E.g., John Regehr et al. started on a Proposal for a Friendly Dialect of C[8]; in their initial posting they proposed some reasonable behaviour for some undefined behaviours and wanted to continue based on finding a consensus among C experts. They eventually gave up[9], partly because finding a consensus is very tough going, and partly because hardware differences make it impractical to make C completely defined in some places. The present paper takes existing, tested programs rather than experts as the yardstick for determining what a piece of C means, and accepts that programs may be hardware-specific. The two examples of disagreements mentioned by Regehr et al. are discussed in the present paper: `memcpy()` in Section 6, and shifts in Section 3.

In his proposal for boringcc,[10] Dan Bernstein partly takes a very similar position to the one I take in the present paper (compile existing programs to behave the same way as old compilers do), and partly defines things beyond that for security reasons, e.g., initializing everything to zero, and tightly defining the behaviour of out-of-bounds accesses (the latter will have a huge performance impact). You can see the present paper as a more elaborate version of the first aspect of boringcc.

Wang et al. [WZKSL13] have written a static program analyzer that tries to find code that an adversarial compiler might optimize away against the intent of the programmer. They identify a C dialect C* that assigns well-defined semantics to code fragments that have undefined behaviour in C; they give some examples of these semantics, but do not discuss them in any depth.

In earlier work [Ert15] I focused on debunking the performance claims of the advocates of adversarial compilation, but I also shortly discussed the intended meaning of programs. Some readers were not satisfied with that discussion, so here I present a more elaborate version of these ideas.

In the Linux kernel/user space interface, the Linux maintainers take exactly the position that I take here for the source-code/compiler interface: Existing programs that worked on an earlier version have to continue working on a new version, even if it can be argued that the programs are buggy.[11]

A debate related to the one that lead to this paper is the one about VAXocentrism[12]. One significant difference is that the VAXocentrism debate was about portability of programs to other platforms, while the present discussion

[8] https://blog.regehr.org/archives/1180
[9] https://blog.regehr.org/archives/1287
[10] https://groups.google.com/forum/#!msg/boring-crypto/48qa1kWignU/o8GGp2K1DAAJ
[11] https://felipec.wordpress.com/2013/10/07/the-linux-way/
[12] http://catb.org/jargon/html/V/vaxocentrism.html

is about whether optimization or newer versions of a compiler on the same platform should compile existing, tested programs to preserve the behaviour of the old versions of these compilers.

Looking at the individual VAXocentric assumptions, it is interesting that some assumptions are now universally valid on general-purpose hardware (flat address space, byte addressing and thus a single pointer format), while advocates of adversarial compilers point to ancient or hypothetical hardware to justify adversarial compilation on modern hardware; other assumptions seem to be on the way to winning (general-purpose machines with big-endian byte order or that trap on unaligned accesses for normal loads/stores are becoming rare); on the other hand, some VAXocentric assumptions (e.g., dereferencing null pointers) are now almost universally invalid on general-purpose platforms.

3 Totally defined C is impractical

Different computer architectures differ in instruction set, but in general have a lot of hardware characteristics in common; however, there are also differences. These (present or former) differences are the source of some prominent undefined behaviours. E.g., the different signed-number representations of earlier times, and resulting differences in signed overflow behaviour led to signed overflow not being defined in C.

While hardware now has consolidated on the 2s-complement representation of signed numbers and all current hardware can perform wraparound arithmetic efficiently, there still are other operations where different hardware produces different behaviour: e.g., shifting by the data width, unaligned accesses, byte order, data sizes, or the behaviour on integer division by zero or overflow.

This is one reason why a total definition of C across all platforms is impractical. E.g., if the totally-defined C specified that shifting by the data width produces 0, the compiler would have to implement shifts more expensively on some machines; and if it specified that it produces the unshifted value, it would have to implement shifts more expensively on other machines. Allowing either result turns this into a portability problem, which is not nice, but far preferable to undefined behaviour. E.g., the rotation idiom `(x<<n)|(x>>(32-n))` produces the intended `x` for `n=0` on either kind of machine, while an adversarial compiler (Clang) compiles this to 0 if it knows that `n=0`.

For shifts the cost of emulating the other behaviour is not that big, but in other cases it can be significant, e.g., the behaviour on unaligned accesses: Simulating unaligned accesses on hardware requiring aligned accesses has significant costs (e.g., 11 instructions instead of 1 for a store on Alpha); conversely, simulating alignment traps on hardware that handles unaligned accesses transparently is not cheap, either, simply because loads and stores are frequent.

In the end, C is a language that is close to the machine, so it is appropriate if differences between machines are reflected in the behaviour of the resulting code in certain cases. Of course, if someone wants to implement a set of compilers for different machines that exhibits common behaviour rather than machine-specific

behaviour for some operations, that is also ok. Such a compiler would not be able to run some programs written for a closer-to-the-machine compiler; that's acceptable, because it's a different compiler, not a newer release of the same compiler.

So the intended behaviour may be hardware-dependent and therefore non-portable. While portability would be nice, this is and always has been an issue that the programmer had to solve, and is outside the scope of the present paper: If a programmer has written a program that only works on, say, AMD64 machines, it should continue to work on AMD64 machines with the next version of the compiler; if it has not worked on ARM before, it is unlikely to work on ARM in the future without changes.

Even portable programs often contain hardware-specific code, often protected by an appropriate `#ifdef`.

4 Basic model of the intended meaning

In the basic model the C compiler maps each C operation consistently to a machine instruction or a sequence of machine instructions *of its choice* that satisfies the requirements of the C standard. E.g., on AMD64, `*p` (where `p` points to a 64-bit value) might be translated to `mov (%rax), %rax`.

This basic mapping is the specification for the behaviour, including non-standard cases. The compiler writer can use the considerable freedom that the C standard provides to choose a behaviour amenable for his goals (performance, safety, compatibility with other compilers, etc.), but once the behaviour has been set, the compiler maintainer must preserve it. The compiler is free to produce more efficient code in optimization modes, and/or in later releases, but has to preserve the behaviour of the basic mapping.

As an example, the AMD64 `mov` instruction above loads the 64-bit value starting at `p` even if `p` is not 8-byte aligned, so on optimization that behaviour has to be preserved. Auto-vectorization is a valid optimization, but has to preserve this behaviour; one way to achieve this is to use the `movdqu`/`vmovdqu` instruction (which does not pose alignment requirements) when autovectorizing a `mov` rather than `movdqa`/`vmovdqa` (which traps when the accessed address is not 16/32-byte aligned).

Even if using `movdqa`/`vmovdqa` was faster (it isn't[13]), using it would be wrong, because it does not preserve behaviour. That does not mean that there should be no way to make use of `movdqa`/`vmovdqa`, only that that must not be achieved by optimizing instructions that do not pose alignment requirements.

5 Loosening the basic model

As explained below, the basic model is a little too tight for a language like C. This means that we have to leave the heights of absolute equivalence, and have to find

[13] http://pzemtsov.github.io/2016/11/06/bug-story-alignment-on-x86.html
http://www.complang.tuwien.ac.at/anton/autovectors/

out whether existing, tested programs rely on certain properties. Fortunately, we have a large corpus of programs for testing that. In some cases, we can also find by reasoning that programs are unlikely to rely on certain properties.

5.1 Out-of-bounds memory accesses

If we want to be absolutely equivalent in the case of out-of-bounds memory accesses, no change is possible (no optimization, and no other change to the executable code). That's because we can use out-of-bounds memory accesses to read the executable machine code, and any change to the executable code can, in theory, change the behaviour.

While there are some programs that access the executable machine code [RS96,PV08], no program relies on the exact contents of specific memory locations. The reason is that such a program would be unmaintainable even without compiler changes: Any change to the source code of such a program, even inserting a piece of code that is not executed, would have a good chance to change its behaviour. Because programs do not rely on that, the compiler can change the machine code, e.g., for optimization.

For data memory, I have seen code that accesses neighbour parameters of a function using address arithmetic.[14] So there are cases where code makes assumptions about the layout of data in memory. These assumptions are probably for data that tend to have the same layout across different compiler versions anyway as long as the definitions of these data and any definitions between them are not changed (e.g., automatic variables in memory or global variables). So a compiler does not have to be extra careful in order not to break these assumptions, the usual amount of care is sufficient.

In addition to programs that rely on a particular result from an out-of-bounds access, there are also programs that perform such an access, but where the actual result is not relied on, because it is not used, the out-of-bounds part is masked away, or because it cancels itself out. Moreover, there are cases where an out-of-bounds address (beyond the one-past-the-end exception) is computed, but not accessed. In such cases an adversarial compiler maintainer feels entitled to compile the program into anything, and such cases have happened (e.g., "optimizing" a finite loop into an infinite loop, or "optimizing" a bounds check away), and of course, a benign compiler will compile the program as intended.

5.2 Uninitialized data

Access to uninitialized data is another issue where absolute equivalence with the basic model would make important optimizations impossible. Consider a variable v at the end of its life (e.g., at the end of a function). Unless the compiler can prove that the location of the variable is not read later as a result of reading uninitialized data (say, reading the uninitialized variable w living in the same

[14] And, to my surprise, this code worked even on RISCs of the day (1990s), despite register-based calling conventions.

location in a different function), v would have to stay in the same location in future compiler versions or other optimization levels; or at least the final value of v would have to be stored in this location, and the initial value of w would have to be fetched from this location.

Fortunately, existing tested programs avoid relying on this equivalence, because it tends to break on maintenance even while keeping the compiler version the same: if the maintenance programmer inserts or deletes a variable or field, the location of v, w, or both can easily change and thus break the program. In practice, compilers have long performed register allocation, copy propagation and other optimizations that can change the value of uninitialized variables, including in compilers that are generally considered benign compilers and have been named as models for boringcc, such as early versions of gcc.

Another way to deal with this issue is to actually initialize data to, say, 0, in the basic model; then rearranging the variables does not change the behaviour of accessing uninitialized variables. This is relatively cheap for automatic scalar variables, because the initialization by the program would turn most of these auto-initializations into dead code, which would be optimized away and therefore cost nothing. For bigger compound data the balance is different: possibly higher initialization cost,[15] and lower register allocation return, so one might consider performing the initialization only for scalars; however, introducing such a difference complicates the programming model and is not programmer-friendly.

5.3 Optimizations

A benign compiler like, say, early gcc, can be viewed as following this looser model. In addition to the optimizations performed by early gcc, many other optimizations are possible in this model that are not covered by early gcc; e.g., auto-vectorization, the pride of recent gcc releases, is compatible with this model.[16] In other words, the compiler does not need to be adversarial in order to perform optimizations.

I have not discussed multi-threading here. It complicates matters, but can probably be resolved with similar reasoning, especially because things like race conditions are so unpredictable that programmers cannot rely on much anyway. However, I do not have enough knowledge in that area to present such a reasoning.

6 Library functions

Many programs do not rely on the implementation internals of library functions, but some do, so it's a good idea to check any potential change in library functions against the corpus of existing programs.

[15] This needs further investigation; loading a cache line from main memory can be much more expensive than zeroing it.

[16] However, I think that it is better to let the programmer express vector operations directly rather than letting the programmer write scalar code, and hoping that he writes it in a way that the compiler can auto-vectorize.

A prominent example is `memcpy()`. If the source and destination ranges do not overlap, the order in which the bytes are loaded and stored does not make a difference, but if they overlap, it does. So an optimization that is in line with the basic model is to check for overlap: If there is overlap, perform something absolutely equivalent to the original implementation; if not, and use the most efficient order.

Looking at the discussion surrounding a change in the `memcpy()` implementation in glibc[17], we can determine something more about what existing, tested programs relied on at the time. Some programs were affected by the difference between the versions, so they called `memcpy()` with overlapping source and destination. Old versions of glibc behave like `memmove()` when the destination address is lower than the source address, and exhibit varying behaviour (that did not lead to such bug reports) when the destination address is higher than the source address; using `memmove()` instead of `memcpy()` did not break the programs, either. But the change that led to the discussion made the programs behave differently than intended, so the programs obviously had overlapping source and destination ranges.

The programs obviously relied on being able to copy from higher to lower addresses even if the ranges overlapped, but did not use `memcpy()` for copying overlapping ranges from lower to higher addresses. So in theory there was the option to make use of this to have a more efficient `memcpy()` than `memmove()` while preserving the intended behaviour. In practice, the performance difference is close to zero, and the use of `memmove()` in glibc 2.24 is a good solution.

7 Conclusion

While the position for adversarial compilers is relatively simple and clear, the intended meaning of programs with undefined behaviour has been called into question. This paper defines a benign position based on preserving the behaviour of existing, tested programs on the platforms that they were tested on. This position does not solve portability problems, but these were problems that C programmers had to solve by themselves already in the good old days of benign compilers.

In the basic model the compiler just maps each C operation to a machine instruction or sequence of machine instructions, consistently, and then preserves that behaviour in optimization or future versions. This basic model is too tight for any optimization, but fortunately, we can loosen this model a little without breaking existing, tested programs. So the results of out-of-bounds accesses and of reading uninitialized data may vary to some extent between compilations, which enables all optimizations available in early (benign) gcc, and also other optimizations, e.g., auto-vectorization.

[17] https://bugzilla.redhat.com/show_bug.cgi?id=638477
https://sourceware.org/bugzilla/show_bug.cgi?id=12518

References

Ert15. M. Anton Ertl. What every compiler writer should know about programmers. In Jens Knoop and M. Anton Ertl, editors, *18. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS'13)*, pages 112–133, 2015. 2

PV08. Gregory B. Prokopski and Clark Verbrugge. Compiler-guaranteed safety in code-copying virtual machines. In *Compiler Construction (CC'08)*, pages 163–177. Springer LNCS 4959, 2008. 5.1

RS96. Markku Rossi and Kengatharan Sivalingam. A survey of instruction dispatch techniques for byte-code interpreters. Technical Report TKO-C79, Faculty of Information Technology, Helsinki University of Technology, May 1996. 5.1

WCC+12. Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. Undefined behavior: What happened to my code? In *Asia-Pacific Workshop on Systems (APSYS'12)*, 2012. 2

WZKSL13. Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. Towards optimization-safe systems: analyzing the impact of undefined behavior. In Michael Kaminsky and Mike Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 260–275. ACM, 2013. 2

Eigenschaften typischer Muster auf geordneten Attributgrammatiken

Christian Berg und Wolf Zimmermann

[1] christian.berg@informatik.uni-halle.de
[2] wolf.zimmermann@informatik.uni-halle.de
Institut für Informatik
Martin-Luther-Universität Halle-Wittenberg

Zusammenfassung. Bei der Implementierung von Spracheigenschaften und statischer Analysen mit Attributgrammatiken treten typische Muster auf. Diese Arbeit stellt typische Muster als Funktion auf Attributgrammatiken vor und untersucht deren Berechenbarkeitseigenschaften. Durch Beweis, dass typische Muster zerlegungserhaltend sind und deren Komposition abgeschlossen ist, wird erreicht, dass nahezu beliebige Attributgrammatiken aus typischen Mustern aufgebaut werden können. Das resultierende Laufzeitverhalten bei der Komposition wird durch die gezeigten Eigenschaften und die Konstruktion von Attributgrammatiken mit typischen Mustern nicht negativ beeinflusst.

1 Einleitung

Die von Knuth in [32] vorgestellten Attributgrammatiken haben sich als geeignet erwiesen Programmiersprachen aber auch Domänen-spezifische Sprachen zu implementieren[36,29,38]. Ein häufiger Kritikpunkt an Attributgrammatiken wurde von Koskimies in [34] wie folgt ausgedrückt:

„The concept of an attribute grammar is too primitive to be nothing but a basic framework, the ‚machine language' of language implementation"

Attributgrammatiken seien also vergleichbar mit Assembler bzgl. der Implementierung von Programmiersprachen. Andererseits existieren mit verschiedenen Erweiterungen, wie Vererbung aus [25], oder Bibliothekssystemen wie [26] und Attributen höherer Ordnung [43], die knappere Beschreibungen ermöglichen. Gemein ist diesen Erweiterungen, dass resultierende Attributgrammatik nicht notwendigerweise berechenbar sein muss. Die Prüfung ob eine beliebige Attributgrammatik berechenbar ist, ist ein schweres Problem[22].

In dieser Arbeit werden typische Muster als Alternative zu eben aufgeführten Erweiterungen vorgestellt. Ausgehend von der Präsentation der Gemeinsamkeiten vieler Spracheigenschaften in Abschnitt 2 und der Vorstellung von Attributgrammatiken und deren Berechenbarkeit in Abschnitt 3 werden typische Muster in Abschnitt 4 eingeführt. Abgeschlossen wird die Arbeit durch den Nachweis der Berechenbarkeit und der Abgeschlossenheit dieser in Abschnitt 5. Eine kurze Zusammenfassung gibt Abschnitt 6.

2 Verwandte Arbeiten

Gegenstand der Arbeiten [3,2,4] ist die Beschreibung einer Domänen-spezifischen Sprache (engl. domain specific language, DSL), die die Steuerung intelligenter Geräte innerhalb eines Gebäudes beschreiben soll. Während in diesen Arbeiten die Semantik nicht formal beschrieben ist, ist eine wichtige Aufgabe die Zuordnung von Aktionen zu Ereignissen auf Basis von Bezeichnern. Die Analyse von Bezeichnern, auch zur Verwendung in der Namensanalyse, wird auch in [13] benötigt bei der Industrieroboter durch eine DSL zur Laufzeit der Programme programmiert werden sollen.

Darüber hinaus existieren aus dem Bereich Domänen-spezifischer Sprachen weitere Beispiele wie [39] – eine Sprache zur Beschreibung der Signalverarbeitung bei der Bitoperationen und Tabellen generiert werden können – oder der Beschreibungssprache für Benutzerschnittstellen in Fahrzeugen aus [20]. Diesen Beispielen und vielen anderen ist gemein, dass eine Namensanalyse zwingend notwendig für weitere Analysen oder die Codegenerierung ist.

Ein häufig in dieser oder ähnlicher Art aufzufindendes Beispiel ist die Codegenerierung oder Analyse auf Basis einer Ausdrucksgrammatik. Beispiel 1.1 (Seite 3 stellt dieses Beispiel unter Ausnutzung der Beschreibung mit Attributgrammatiken vor.

Attributgrammatiken zur Sprachspezifikation wurden von Knuth in [32] eingeführt. Aufgrund der Komplexität bei der Bestimmung der Berechenbarkeit – siehe [22] – dieser sehr allgemeinen Attributgrammatiken wurden eingeschränkte Attributgrammatiken. Eine mögliche Einschränkung stellen Attributgrammatiken mit fester Besuchsstrategie dar, wie diese in [35,23] vorgestellt wurden, oder Attributgrammatiken mit statisch bzw. teilweise statisch bestimmter Berechnungsstrategie, wie diese in [27] oder [31] vorgestellt wurden. Die von Kastens in [27] vorgestellten Attributgrammatiken wurden bereits in vielen Arbeiten und Werkzeugen herangezogen bzw. verwendet[29,30,10,41,28].

Die häufig erwähnte Kritik an Attributgrammatiken – zu hoher Spezifikationsumfang – führte zu einer Reihe von Erweiterungen mit dem Ziel diese Komplexität zu verringern, wie bspw. entfernten Attributen oder verschiedenen Arten der Termersetzung, siehe u. a. [18,11,15,17,40,43]. Boyland beschreibt in [11], dass für entfernte Attribute die Bestimmung der Berechenbarkeit nicht nur schwer (d. h. algorithmisch exponentielle Laufzeit erfordert) sondern sogar unentscheidbar ist. Arbeiten zur Termersetzung, wie bspw. [43,40] lassen sich durch einfache Konventionen innerhalb des Gerüsts der geordneten Attributgrammatiken anwenden, für andere Ansätze, wie [15] ist keine Arbeit bekannt die solche Eigenschaften untersucht.

Einfache Ansätze wie Vererbung oder Modulsysteme sowie einfache Methoden der Textersetzung wie [29,38,25] basieren darauf, dass das Resultat im Grunde vollständig neu betrachtet werden muss, somit sind potentielle Garantien vor Kombination nicht nutzbar.

```
1  -- Namensanalyse
2  rule Root ::= Expr
3  attr Expr.envIn = []
4
5  rule Expr ::= VarDef Expr Expr
6  attr Expr2.envIn = Expr1.envIn
7       Expr3.envIn = ((VarDef.name, Expr2.value):Expr1.envIn)
8       cond VarDef.name ∉ Expr1.envIn =>error "Already defined " ++ VarDef.name
9
10 rule Factor ::= VarRef
11 attr Factor.value = Factor.envIn[VarRef.name]
12      cond VarRef.name ∉ Factor.envIn =>error "Unknown Variable " ++ VarRef.name
13
14 -- Konstantenberechnung
15 rule Root ::= Expr
16 attr report "Output value = " ++ Expr.value
17
18 rule Term ::= Term Factor
19 attr Term1.value = Term2.value * Factor.value
20
21 rule Expr ::= Expr Term
22 attr Expr1.value = Expr2.value + Term.value
23
24 rule Expr ::= VarDef Expr Expr
25 attr Expr1.value = Expr3.value
26
27 -- Identitätsausgabe
28 rule Root ::= Expr
29 attr report "Identity Code = " ++ Expr.output
30
31 rule Expr ::= VarDef Expr Expr
32 attr Expr1.output = "let " ++ VarDef.name ++ " = " ++ Expr2.output ++
33                     " in " ++ Expr3.output
34
35 rule Term ::= Term Factor
36 attr Term1.output = Term2.output ++ " * " ++ Factor.output
37
38 rule Expr ::= Expr Term
39 attr Expr1.output = Expr2.output ++ " + " ++ Term.output
```

In den ersten 12 Zeilen dieses Beispiels wird der semantisch relevante Teil der Namensanalyse präsentiert, indem eine Art Definitionstabelle in Haskell-artiger Syntax leer initialisiert wird (Zeile 3) und mit den Bezeichnern, denen mittels **`let`**-Ausdrücken Werte zugewiesen werden (Zeile 7), die Werte aus dieser Definitionstabelle zur weiteren Berechnung weiter verwendet werden (Zeile 11). Semantisch geprüft werden Definition und Benutzung (Zeilen 8 und 12). Im folgenden Abschnitt von Zeile 14 bis 26 wird eine Art Konstantenfaltung durchgeführt und das Ergebnis dieser ausgegeben. Eine der Codegenerierung angelehnte Identitätsausgabe erfolgt in den letzten Zeilen des Beispiels.
Der Umfang des vollständigen Beispiels gemeinsam mit der Definition der benötigten Attribute und Definition der Grammatik umfasst über 110 Zeilen mit mehrheitlich Kopieranweisungen. Werden die in [38] und [29] vorgestellten Paradigmen angwandt, verringert sich der Quellumfang auf knapp unter 100 Zeilen. Die Differenz beträgt in diesem Fall weniger als 20 Zeilen.

Beispiel 1.1 – Semantisch relevanter Teil der Attributierungsregeln zur Ausgabegenerierung, Namensanalyse und Wertberechnung für eine Ausdrucksgrammatik

3 Attributgrammatiken

Grundlage für Attributgrammatiken sind kontextfreie Grammatiken. Kontextfreie Grammatiken lassen sich als (erweiterte) Backus-Naur-Form[5] ((E)BNF) oder in Form von Syntaxdiagrammen (siehe z. B. [44,45,8]) darstellen.

Definition 1. *Ein Tupel* $G \triangleq (N, T, P, Z)$ *wobei*

- N *eine endliche Menge an Nichtterminalen,*
- T *eine endliche Menge Terminale,*
- $P \subseteq N \times (N \cup T)^*$ *die Menge der Produktionen und*
- $Z \in N$ *dem (ausgezeichnetem)* Startsymbol

heißt ***kontextfreie Grammatik*** *genau dann, wenn* $N \cap T = \emptyset$

Die Menge der Produktionen induziert eine Ableitungsrelation. Ist $p \in P$ mit $p\colon X ::= u\ Y\ v$ für $X \in N$, $Y \in \Sigma, u, v \in \Sigma^*$ und $\Sigma = N \cup T$; dann ist Y aus X *ableitbar*, geschrieben als $X \Rightarrow Y$. Für die Relation der Ableitbarkeit sind transitiver und reflexiv-transitiver Abschluss wie üblich definiert.

Folgende Definition von Attributgrammatiken folgt der Darstellung aus [32] und [44]. Dies gilt ebenso für die darauf folgenden.

Definition 2. *Eine* ***attributierte Grammatik*** *ist ein Tupel* $AG \triangleq (G, A, R, B)$*, wobei*

- $G \triangleq (N, T, P, Z)$ *eine kontextfreie Grammatik ist, die eine abstrakte Syntax definiert,*
- $A \triangleq \biguplus_{X \in \Sigma} A_X$ *eine endliche Menge von* ***Attributen****,*
- $R \triangleq \biguplus_{p \in P} R_p$ *eine endliche Menge* ***Attributierungsregeln*** *der Form* $a_0 \leftarrow f(a_1, \cdots, a_k)$*, wobei* f *eine Funktion ist und*
- $B \triangleq \biguplus_{p \in P} B_p$ *eine endliche Menge von* ***Bedingungen*** *der Form* $\varphi(a_1, \cdots, a_k)$*, wobei* φ *ein Prädikat ist.*

Für alle $a_0 \leftarrow f(a_1, \cdots, a_k) \in R_p$ *einer Produktion* $p \in P, p\colon \mathtt{X}_0 ::= \mathtt{X}_1 \cdots \mathtt{X}_n$ *muss gelten:* $a_i \in A_{\mathtt{X}_0} \cup \cdots A_{\mathtt{X}_n}, i = 0, \cdots, k$ *und für alle* $\varphi(a_1, \cdots, a_k) \in B_p$ *für eine Produktion* $p \in P, p\colon \mathtt{X}_0 ::= \mathtt{X}_1 \cdots \mathtt{X}_n$ *muss gelten:* $a_i \in A_{\mathtt{X}_0} \cup \cdots A_{\mathtt{X}_n}, i = 1, \cdots, k$*.*

Die Definition von Attributgrammatiken nach dieser Art erlaubt allerdings schnell inkonsistente oder nicht berechenbare Attributgrammatiken, sodass folgende Definitionen notwendig sind.

Definition 3. *Eine Attributgrammatik* $AG \triangleq (G, A, R, B)$ *heißt* ***konsistent*** *genau dann, wenn für alle abstrakten Syntaxbäume* B *folgende Bedingungen erfüllt sind:*

- *für jedes Attribut* $a \in A$ *in jedem Knoten* k *höchstens eine definierende Regel* $\mathtt{X}.a \leftarrow e \in R$ *existiert und*
- *der* $Typ(k) = \mathtt{X}$ *ist.*

Die Attributgrammatik heißt ***vollständig****, wenn in jedem Knoten k mindestens eine Regel $\mathtt{X}.a \leftarrow e \in R$ existiert.*

In dieser Definition sind Knoten K vom Typ X genau dann, wenn es eine Regel in der abstrakten Syntax $G \triangleq (N, T, P, Z)$ existiert, sodass zum Aufbau des Knotens K im abstrakten Syntaxbaum T eine Produktion mit linker Seite X angewendet wurde. Technische Details zum Aufbau abstrakter Syntaxbäume sind [1,24] zu entnehmen.

Ziel dieser Arbeit ist der Nachweis, dass typische Muster abgeschlossen für verschiedene Arten der Berechenbarkeit von Attributgrammatiken sind. Dafür ist die Definition dieser Arten der Berechenbarkeit notwendig. Um für eine Attributgrammatik und eine potentielle Eingabe, d. h. einen abstrakten Syntaxbaum, zu bestimmen ob diese zusammen berechenbar sind, hilft folgende Definition:

Definition 4. *Für eine attributierte Grammatik $AG \triangleq (G, A, R, B)$ mit abstrakter Syntax $G \triangleq (N, T, P, Z)$ und einem abstrakten Syntaxbaum T von G mit Knoten $K_0, \cdots, K_n$ heißt eine Liste $[K_{i_0}.a_0, \cdots, K_{i_m}.a_m]$ aller Attribute aller Knoten von T* ***berechenbare Reihenfolge*** *genau dann, wenn jedes Attribut jedes Knotens genau einmal in der Liste vorkommt und für jedes Attribut $K_{i_j}.a_j$ eine Definition $K_{i_j} \leftarrow f(K_{i_{j_1}}.a_{j_1}, \cdots, K_{i_{j_q}}.a_{j_q})$ der entsprechenden Produktion p der Attributgrammatik existiert, sodass $j_1, \cdots, j_q < j$ ist.*

Implizit ist in der Definition der berechenbaren Reihenfolge bereits der Begriff der Abhängigkeit von Attributen enthalten:

Definition 5. *Sei $AG \triangleq (G, A, R, B)$ eine attributierte Grammatik mit abstrakter Syntax $G \triangleq (N, T, P, Z)$. Für jede Produktion $p\colon \mathtt{X}_0 ::= \mathtt{X}_1 \cdots \mathtt{X}_n \in P$ mit Attributierungsregel $\mathtt{X}_j.b \leftarrow f(\cdots \mathtt{X}_i.a \cdots)$ heißt $\mathtt{X}_i.a \rightarrow \mathtt{X}_j.b \subseteq A \times A$* ***lokale Abhängigkeit der Produktion*** *p. Für eine Produktion $p \in P$ heißt $DG_p \triangleq (A_p, DDP_p)$ mit $A_p \triangleq A_{\mathtt{X}_0} \cup \cdots \cup A_{\mathtt{X}_n}$ und $DDP_p \triangleq \{\mathtt{X}_i.a \rightarrow \mathtt{X}_j.b : \mathtt{X}_j.b \leftarrow f(\cdots \mathtt{X}_i.a \cdots) \in R_p\}$* ***lokaler Abhängigkeitsgraph zu*** *p.*

Auf Basis des Abhängigkeitsgraphen von Produktionen lassen sich erste berechenbare Klassen von Attributgrammatiken definieren.

Definition 6. *Eine attributierte Grammatik $AG \triangleq (G, A, R, B)$ mit abstrakter Syntax $G \triangleq (N, T, P, Z)$ heißt* ***lokal azyklisch*** *genau dann, wenn für alle Produktionen $p \in P$ die Graphen DG_p nach Definition 5 azyklisch sind.*

Definition 5 und Definition 6 führen zu der Definition wohldefinierter Attributgrammatik – jenen Attributgrammatiken für die eine berechenbare Reihenfolge (siehe Def. 4) existiert.

Definition 7. *Eine konsistente attributierte Grammatik AG heißt* ***wohldefiniert****, genau dann, wenn für jeden abstrakten Syntaxbaum T eine berechenbare Reihenfolge existiert.*

Lemma 1. *([44,42]) Eine konsistente attributierte Grammatik AG ist genau dann wohldefiniert, wenn sie vollständig ist und für jeden abstrakten Syntaxbaum T der Abhängigkeitsgraph DT_T azyklisch ist.*

Durch die Bestimmung in welcher Reihenfolge Attribute auswertbar sind, lassen sich in der Praxis effiziente Evaluatoren und Codegeneratoren für Attributgrammatiken implementieren.

Definition 8. *Für eine attributierte Grammatik* $AG \triangleq (G, A, R, B)$ *mit abstrakter Syntax* $G \triangleq (N, T, P, Z)$ *und allen Symbolen* $\mathtt{X} \in \Sigma$ *heißt die Partitionierung von* $A_{\mathtt{X}} = A_{\mathtt{X}}(1) \uplus \cdots \uplus A_{\mathtt{X}}(m_{\mathtt{X}})$ ***zulässige Zerlegung*** *genau dann, wenn für alle Symbole* $\mathtt{X}$ *gilt* $A_{\mathtt{X}}(i) \subseteq AS_{\mathtt{X}}$ *für* $i = m, m-2, \cdots$ *und* $A_{\mathtt{X}}(i) \subseteq AI_{\mathtt{X}}$ *für* $i = m-1, m-3, \cdots$.

Definition 9. *Eine attributierte Grammatik* $AG \triangleq (G, A, R, B)$ *mit abstrakter Syntax* $G \triangleq (N, T, P, Z)$ *heißt* ***zerlegbar*** *genau dann, wenn sie lokal azyklisch ist und für jedes Symbol* $\mathtt{X} \in \Sigma$ *eine zulässige Zerlegung* $A_{\mathtt{X}} = A_{\mathtt{X}}(1) \uplus \cdots \uplus A_{\mathtt{X}}(m_{\mathtt{X}})$ *existiert, sodass für jeden abstrakten Syntaxbaum* T *eine berechenbare Reihenfolge existiert so, dass die Attribute des Knoten* K *mit* $Typ(K) = \mathtt{X}$ *in der Reihenfolge* $A_{\mathtt{X}}(1), \cdots, A_{\mathtt{X}}(m_{\mathtt{X}})$ *berechnet werden können.*

Definition 10. *Eine zerlegbare attributierte Grammatik* $AG \triangleq (G, A, R, B)$ *mit zulässiger Zerlegung* $A_{\mathtt{X}} = A_{\mathtt{X}}(1) \uplus \cdots \uplus A_{\mathtt{X}}(m_{\mathtt{X}})$ *für alle Symbole* $\mathtt{X} \in \Sigma$ *heißt* ***zerlegt****.*

Eine Art von Attributgrammatiken bei der bereits ein Großteil zur Berechenbarkeit vor Aufbau des abstrakten Syntaxbaums erzeugt werden kann ist folgende:

Definition 11. *([31]) Sei* $AG \triangleq (G, A, R, B)$ *eine attributierte Grammatik mit abstrakter Syntax* $G \triangleq (N, T, P, Z)$*. Für alle Produktionen* $p \in P$ *mit* $p : \mathtt{X}_0 ::= \mathtt{X}_1 \cdots \mathtt{X}_n$ *heißt* $NDDP_p = DDP_p^+ \setminus \{(\mathtt{X}_i.a, \mathtt{X}_j.b) : \mathtt{X}_i.a \text{ und } \mathtt{X}_j.b \text{ werden in } p \text{ definiert }\}$ ***normalisierte direkte Abhängigkeit der Produktion*** p*. Für alle Symbole* $\mathtt{X} \in \Sigma$ *sind die* ***induzierten Attributabhängigkeiten*** *das kleinste System von Mengen* $IDS_{\mathtt{X}} \subseteq A_{\mathtt{X}} \times A_{\mathtt{X}}$, $IDP_p \subseteq A \times A$, *welches folgende Gleichungen erfüllt:*

1. $NDPP_p \subseteq IDP_p$
2. $IDS_{\mathtt{X}} = \{\mathtt{X}.a \to \mathtt{X}.b \colon \exists q \in P \text{ sodass } \mathtt{X}.a \to \mathtt{X}.b \in IDP_q^+\}$
3. $IDP_p = IDP_p \cup IDS_{\mathtt{X}_0} \cup \cdots \cup IDS_{\mathtt{X}_n}$

IDS sind die ***induzierten Abhängigkeiten zwischen Symbolattributen****, IDP sind die* ***induzierten Abhängigkeiten zwischen Attributvorkommen****.*

Die Attributgrammatik AG heißt ***absolut azyklisch*** *genau dann, wenn für alle Nichtterminale* $\mathtt{X} \in N$ *und alle Produktionen* $p \in P$ $IDS_{\mathtt{X}}$ *und* IDP_p *azyklisch sind.*

Diese Definition legt implizit eine Halbordnung der Attribute fest. Kann in jedem Kontext eines Symbols einer Attributgrammatik für die mit diesem Symbol assoziierten Attribute eines Auswertereihenfolge, d.h. Zerlegung, angegeben werden, die diese Halbordnung enthält, dann heißt diese Attributgrammatik *geordnet.*

Definition 12. *Sei* $AG \triangleq (G, A, R, B)$ *eine attributierte Grammatik mit abstrakter Syntax* $G \triangleq (N, T, P, Z)$*. Für alle Symbole* $\mathtt{X} \in \Sigma$ *seien*

1. $T_{\mathtt{X}}(-1) \triangleq \emptyset$
2. $T_{\mathtt{X}}(0) \triangleq \emptyset$
3. $T_{\mathtt{X}}(2k-1) \triangleq \{a \in AS_{\mathtt{X}}$ *so, dass* $\forall b \in A_{\mathtt{X}} : \mathtt{X}.a \to \mathtt{X}.b \in IDS_{\mathtt{X}} \Rightarrow \exists j \leq 2k-1$ *mit* $\mathtt{X}.b \in T_{\mathtt{X}}(j)\}$
4. $T_{\mathtt{X}}(2k) \triangleq \{a \in AI_{\mathtt{X}}$ *so, dass* $\forall b \in A_{\mathtt{X}} : \mathtt{X}.a \to \mathtt{X}.b \in IDS_{\mathtt{X}} \Rightarrow \exists j \leq 2k$ *mit* $\mathtt{X}.b \in T_{\mathtt{X}}(j)\}$
5. $A_{\mathtt{X}}(i) \triangleq T_{\mathtt{X}}(m-i+1) \setminus T_{\mathtt{X}}(m-i-1)$ *mit* $i = 1, \cdots, m$

wobei $m \in \mathbb{N}$ *minimal und* $T_{\mathtt{X}}(m-1) \cup T_{\mathtt{X}}(m) = A_{\mathtt{X}}$*. Die attributierte Grammatik heißt* ***geordnet*** *(OAG) genau dann, wenn sie mit den Partitionen* $A_{\mathtt{X}} = A_{\mathtt{X}}(1) \uplus \cdots \uplus A_{\mathtt{X}}(m_{\mathtt{X}})$ *zerlegt ist und für alle Produktionen* $p \in P$ *der erweiterte Abhängigkeitsgraph* $EDP_p \triangleq IDP_p \cup \{\mathtt{X}.a \to \mathtt{X}.b : \exists h, k \in \mathbb{N}$ *mit* $\mathtt{X}.a \in A_{\mathtt{X}}(h) \wedge \mathtt{X}.b \in A_{\mathtt{X}}(k) \wedge h < k\}$ *azyklisch ist.*

Kastens zeigt in [27] wie durch Hinzufügen zusätzlicher Abhängigkeiten jede zerlegbare Attributgrammatik in eine geordnete Attributgrammatik überführt werden kann.

Im Allgmeinen ist die Kombination zweier geordneter Attributgrammatiken über derselben abstrakten Syntax nicht notwendigerweise wieder geordnet[37,27,12].

Ziel dieser Arbeit ist es einen Formalismus vorzustellen, der jedoch genau dies ermöglicht.

4 Typische Muster und deren Äquivalenzen

Ein typisches Muster ist eine Funktion im Raum der Attributgrammatiken über eine abstrakte Syntax.

Definition 13. *Sei* $G \triangleq (N, T, P, Z)$ *eine abstrakte Syntax und* $\mathsf{AG}_G \triangleq (G, \mathsf{A}, \mathsf{R}, \mathsf{B})$ *die Menge aller Attributgrammatiken mit abstrakter Syntax* G*. Eine Funktion* $\mathcal{M}_G \colon \mathsf{AG}_G \to \mathsf{AG}_G$ *heißt* ***typisches Muster*** *über der abstrakten Syntax* G *genau dann, wenn für alle zerlegbaren Attributgrammatiken* $AG \in \mathsf{AG}_G$ *die resultierende Attributgrammatik* $AG' = \mathcal{M}_G(AG)$ *zerlegbar ist.*

Die Definition dieser Muster lässt sich durch Elementweise Definition der Transferfunktionen über den veränderten Mengen erreichen, sodass $\mathcal{M}_G \triangleq (\mathcal{M}_{G,A}, \mathcal{M}_{G,R}, \mathcal{M}_{G,B})$ verwendet werden kann. Zur einfacheren Argumentation werden die Bedingungen von Attributgrammatiken nicht beachtet. Diese lassen sich durch Attribute emulieren[44]. Die praktische Bedeutung solcher Bedingungen liegt in der Möglichkeit diese „Attribute“ als Abbruchkriterium bspw. der Übersetzung zu interpretieren. Im weiteren werden daher nur die beiden Unterfunktionen $\mathcal{M}_{G,A}$ und $\mathcal{M}_{G,R}$ betrachtet. Im folgenden wird auf die Verwendung des Index G verzichtet da dieser konstant ist und aus dem Kontext klar ist.

Die Änderung einer Attributgrammatik im Sinne der Transformation durch Musteranwendung lässt sich im Allgemeinen durch verschiedene Kombinationen des Hinzufügens und Entfernen von Attributen bzw. Attributierungsregeln darstellen. Somit können diese „Transferfunktionen“ über Attributgrammatiken wie folgt definiert werden.

Lemma 2. *Sei* $AG \in \mathsf{AG}, AG \triangleq (G, A, R, B)$ *eine zerlegbare Attributgrammatik. Für jede solche Attributgrammatik existiert ein Muster* $\mathcal{M} \triangleq (\mathcal{M}_A, \mathcal{M}_R)$ *sodass die resultierende Attributgrammatik* $AG' = (G, \mathcal{M}_A(A), \mathcal{M}_R(R), B)$ *zerlegbar ist, wobei*

- $\mathcal{M}_A(A) = (A \setminus \mathcal{M}_{A,-}(A)) \cup \mathcal{M}_{A,+}(A)$ *und*
- $\mathcal{M}_R(R) = (R \setminus \mathcal{M}_{R,-}(R)) \cup \mathcal{M}_{R,+}(R)$

ist.

Beweis. Durch Angabe eines Musters: Sei $\mathcal{M}_{A,-}(A) = \emptyset$, $\mathcal{M}_{R,-}(R) = \emptyset$ *und* $\mathcal{M}_{A,+}(A) = \emptyset$ *und* $\mathcal{M}_{R,+}(R) = \emptyset$. *Sei die resultierende Attributgrammatik nicht zerlegbar, dann steht dies im Widerspruch zur Vorraussetzung.*

Der Beweis zu Lemma 2 stellt eine triviale Lösung dar. Diese Lösung – keine Änderung der Attributgrammatik – steht immer zur Verfügung. Für eine praktisch relevante Beschreibung von Mustern müssen diese unabhängig von der konkreten Attributgrammatik, auf der diese angewendet werden, beschreibbar sein.

Zur Beschreibung eines Musters unabhängig von einer kontextfreien Grammatik bzw. unabhängig von der Attributgrammatik auf die Muster angewendet werden sollen ist es notwendig folgende Definition aufzustellen:

Definition 14. *Seien* a_i, $0 \leq i \leq n$, *für* $n \in \mathbb{N}$ *Variablen der Sorte* $\mathfrak{A}$, *sogenannte Musterattribute, dann sind* ***Musterattributwertterme*** $t_{[\![\mathfrak{A}]\!]}$ *wie folgt induktiv definiert:*

- c *ist ein Musterattributwertterm, wobei* c *eine Konstante ist;*
- a_i *ist ein Musterattributwertterm und*
- *für* $t^{(1)}_{[\![\mathfrak{A}]\!]}, \cdots, t^{(k)}_{[\![\mathfrak{A}]\!]}$, $k \in \mathbb{N}$ *Musterattributwertterme ist* $f(t^{(1)}_{[\![\mathfrak{A}]\!]}, \cdots, t^{(k)}_{[\![\mathfrak{A}]\!]})$ *ein Musterattributwertterm wobei* f *eine beliebige Funktion ist.*

Aus der funktionalen Programmierung bekannt ist, dass beliebige k-stellige Funktionen durch eine Funktion mit genau einem Argument darstellbar sind. Im folgenden werden daher Funktionen mit mehreren Argumenten wie in der Literatur üblich dargestellt ohne dabei auf Currying zurückzugreifen. Zu diesem Verfahren siehe auch [6,21].

Ausgehend von Musterattributwerttermen lassen sich die eigentlichen Musterterme beschreiben, dabei werden diese Terme in eine Musterattributierungsregel eingebettet und bilden gemeinsam einen Musterattributierungsterm.

Definition 15. *Sei $t_{[\![\mathfrak{A}]\!]}$ eine Musterattributwertterm über Musterattribute a_i, wobei $0 \leq i \leq n$, $n \in \mathbb{N}$. Darüber hinaus seien r_j, $0 \leq j \leq m$, $m \in \mathbb{N}$, Variablen der Sorte $\mathfrak{R}$, sogenannte Musterregeln, dann heißt eine Attributierungsregel*

$$\textbf{rule } r_j \textbf{ attr } a_i \leftarrow t_{[\![\mathfrak{A}]\!]}$$

***Musterattributierungsterm** genau dann, wenn a_i nicht in $t_{[\![\mathfrak{A}]\!]}$ vorkommt.*

Die Abbildung dieser Terme auf konkrete Attributgrammatiken lässt sich mittels Prädikaten und Abbildungsrelationen erreichen. Die Idee hinter der Verwendung von Prädikaten ist, dass durch Anwendung eines Prädikats die Beschreibung der Abbildung vergleichsweise unabhängig gestaltet werden kann.

Definition 16. *Sei* **rule** r_j **attr** $a_i \leftarrow t_{[\![\mathfrak{A}]\!]}$ *ein Musterattributierungsterm mit r_j, $0 \leq j \leq m, m \in \mathbb{N}$ Variablen der Sorte $\mathfrak{R}$ und a_i Variablen der Sorte $\mathfrak{A}$ mit $0 \leq i \leq n, n \in \mathbb{N}$; $t_{[\![\mathfrak{A}]\!]}$ einem Musterattributwertterm. Für eine Attributgrammatik $AG \triangleq (G, A, R, B)$ mit abstrakter Syntax $G \triangleq (N, T, P, Z)$ heißt das Prädikat $p_{AG}(r_j, a_i, t_{[\![\mathfrak{A}]\!]}, q)$ **abbildendes Prädikat** genau dann, wenn für alle Produktionen $q \in P$ der Attributgrammatik das Prädikat zu wahr oder falsch auswertet und $\bar{p}_{AG}(r_j, a_i, t_{[\![\mathfrak{A}]\!]}, q)$ **löschendes Prädikat** analog. Dabei gilt, wird p zu wahr ausgewertet, dann impliziert dies, dass $\bar{p}$ nicht zu wahr ausgewertet wird.*

Intuitiv klar ist, dass bei zwei Prädikaten für eine Produktion q zusammen mit der letzten Bedingung von Definition 16 genau drei Kombinationen möglich sind. Eine Produktion kann dann abgebildet, gelöscht oder ignoriert werden. Letzteres würde somit beschreiben, dass ein Musterattributierungsterm nicht auf die Produktion abgebildet wird. Im Falle dass das löschende Prädikat zu wahr auswertet wird die abgebildete Attributierungsregel entfernt bzw. ersetzt.

Durch die Prädikate werden somit die „passenden" Produktionen für ein Muster ausgewählt. Das eigentliche Hinzufügen (oder Entfernen) von Attributierungsregeln und Attributen mittels Substitution der Musterattribute durch die in der Attributgrammatik vorkommenden Attribute[3]. Dabei können Musterattribute auch mit dem „leeren Attribut" substituiert werden, sodass dieses Musterattribut entfernt wird.

Definition 17. *Sei $\mathfrak{R}$ die Menge aller Musterattributierungsterme $t_{j,[\![\mathbb{R}]\!]}$ der Form* **rule** r_j **attr** $a_i \leftarrow t_{[\![\mathfrak{A}]\!]}$ *für $0 \leq i \leq n, 0 \leq j \leq m$ für $m, n \in \mathbb{N}$. Für eine zerlegbare Attributgrammatik $AG \triangleq (G, A, R, B)$ und abstrakter Syntax $G \triangleq (N, T, P, Z)$ mit abbildenden und löschenden Prädikaten $p_{AG}(r_j, a_i, t_{[\![\mathfrak{A}]\!]}, q)$ und $\bar{p}_{AG}(r_j, a_i, t_{[\![\mathfrak{A}]\!]}, q)$ für Produktionen $q \in P$ der Attributgrammatik heißt die Menge $\mathfrak{R}'_{AG} \triangleq \{r_j \sigma_{r_j,q} \colon \sigma_{r_j,q} \left[\mathtt{a}_1/a_1 \cdots \mathtt{a}_n/a_n\right], r_j \in \mathfrak{R}, q \in P, p_{AG}(r_j, a_i, t_{[\![\mathfrak{A}]\!]}, q)$ werte zu wahr aus und $\bar{p}_{AG}(r_j, a_i, t_{[\![\mathfrak{A}]\!]}, q)$ zu falsch $\}$ **auf** AG **abgebildete Musterattributierungsterme**. Das Attribut $\mathtt{a}_k$ heißt* **in** r_j **abgebildete Attribut** *genau dann, wenn in einer Substitution $\sigma_{r_j,q}$ $\mathtt{a}_k/a_k$ enthalten ist.*

[3] Hierbei werden praktisch zwei Substitutionen angewendet. Zur einfacheren Präsentation beschränkt sich die Präsentation jedoch auf die Substitution der Attribute.

Halten abgebildete Musterattributierungsterme eine Reihe von Eigenschaften ein, so lässt sich zeigen, dass deren Anwendung auf eine Attributgrammatik der Anwendung eines Musters entspricht. Folgende Definition legt die dafür notwendigen Eigenschaften fest:

Definition 18. *Seien $\mathfrak{R}$ die Menge aller Musterattributierungsterme und $\mathfrak{R}'_{AG}$ die Menge aller abgebildeten Musterattributierungsterme für eine zerlegbare Attributgrammatik $AG \triangleq (G, A, R, B)$ mit abstrakter Syntax $G \triangleq (N, T, P, Z)$ und den Substitutionen $\sigma_{r_j,q}$ für $q \in P$, heißt eine Substitution **setzend** genau dann wenn für $a_i \leftarrow t_{[\![\mathfrak{A}]\!]}$ das abgebildete Attribut $\mathtt{a}_i$ sonst nicht in $\sigma_{r_j,q}$ vorkommt und eine der folgende Bedingungen für bestehende Attributierungsregeln $r \in R_q$ der Form* **rule**$\mathtt{X} ::= u$ **attr**$_i \leftarrow e$*,gilt:*

- $\mathtt{a}_i$ *kommt nicht im Ausdruck e vor und alle abgebildeten Attribute* $\mathtt{a}_k$ *in* $\sigma_{r_j,q}$ *kommen bereits in e vor oder*
- $\mathtt{a}_i$ *kommt nicht im Ausdruck e vor und alle abgebildeten Attribute* $\mathtt{a}_l$ *in* $\sigma_{r_j,q}$ *die nicht bereits in e vorkommen sind nicht in A enthalten*

oder keine Attributierungsregel für $\mathtt{a}_i$ *in* R_q *existiert.*

Mit Hilfe dieser Eigenschaften lässt sich nun zeigen, dass solche Musterattributierungsterm, die abgebildeten Musterattributierungsterme und setzende Substitutionen ein Muster auf einer Attributgrammatik definieren.

5 Eigenschaften Typischer Muster

Theorem 1. *Sei $AG \triangleq (G, A, R, B)$ mit abstrakte Syntax $G \triangleq (N, T, P, Z)$ eine zerlegbare Attributgrammatik und $\mathfrak{R}$ eine Menge von Musterattributierungstermen mit setzenden Substitutionen $\sigma_{r_j,q}$ für alle $q \in P$, so definiert $\mathfrak{R}'_{AG}$ ein Muster.*

Beweis. Mittels Konstruktion der resultierenden Attributgrammatik
$AG' \triangleq (G, A', R', B)$ mit

- $A'_q = A_q \cup A_{q,+}$ *wobei* $A_{q,+} = \{\mathtt{a}_{k,j} \colon \exists r_j \in \mathfrak{R}, \mathtt{a}_k$ *abgebildetes Attribut in* $\sigma_{r_j,q}$ *und* $\mathtt{a}_k \notin A\}$
- $R'_q = (R_q \setminus R_{q,-}) \cup R_{q,+}$

wobei

$$R_{q,-} = \{r \colon r \in R_q \wedge r : \textbf{rule } r \textbf{ attr } \mathtt{a}_i \leftarrow e \wedge \exists r_j \in \mathfrak{R} \colon \mathtt{a}_i \text{ abgebildetes Attribut in } \sigma_{r_j,q}\}$$
$$R_{q,+} = \{r_j \sigma_{r_j,q} \colon r_j \in \mathfrak{R}, q \in P\}$$

und Nachweis der Zerlegbarkeit für die resultierende Attributgrammatik AG'.

Angenommen AG' sei nicht zerlegbar, dann ist, ohne Beschränkung der Allgemeinheit in q eine der folgenden Bedingungen verletzt:

1. *der lokale Abhängigkeitsgraph für q ist zyklisch oder*
2. *die Zerlegung für eines der in q vorkommenden Symbole nicht zulässig ist.*

Fall 1 kann nach Definition 18 nur eintreten, wenn AG bereits nicht zerlegbar war und steht somit im Widerspruch zur Voraussetzung; es werden höchstens neue Attribute einer bestehenden Attributierungsregel als zusätzliche Abhängigkeit mitgegeben.

Sei $\mathtt{X} \in (N \cup T)$ das Symbol für die die Zerlegung nicht zulässig ist. Es gilt wieder verschiedene Fälle zu betrachten:

1. *$\mathtt{a}_i$ ist ein abgebildetes Attribut, sodass $\mathtt{a}_i \in A_+$ – der Fall der hinzugefügten Attributierung oder*
2. *$\mathtt{a}_i$ ist das abgebildetes Attribut zu a_i mit zugeordneter Attributierungsregel* **rule** r_j **attr**$a_i \leftarrow t_{[\![\mathfrak{A}]\!]}$, *sodass $q \in R_{q,-}$ und $r_j\sigma_{r_j,q} \in R_{q,+}$ gilt – der Fall der Ersetzung einer Attributierungsvorschrift,*

Ohne Beschränkung der Allgemeinheit sei der maximale Partitionierungsindex in AG für $\mathtt{X}$ $m_\mathtt{X}$ mit der Partitionierung $A_\mathtt{X} = A_\mathtt{X}(1) \uplus \cdots \uplus A_\mathtt{X}(m_\mathtt{X})$. Im ersten Fall wird $m_\mathtt{X}$ um 2 erhöht, sodass für synthetisierte Attribute $\mathtt{a}_i \in A_\mathtt{X}(m_{\mathtt{X}+2})$ gilt und $A_\mathtt{X}(m_{\mathtt{X}+1}) = \emptyset$ und für ererbte Attribute $\mathtt{a}_i \in A_\mathtt{X}(m_{\mathtt{X}+1})$ sowie $A_\mathtt{X}(m_{\mathtt{X}+2}) = \emptyset$ gilt. Ist die Partitionierung für AG' dann nicht zulässig, so muss dies bereits in AG der Fall gewesen sein, wiederum im Widerspruch zur Voraussetzung.

Nach Definition 18 werden höchstens noch nicht bestehende Attribute bei der Ersetzung den Abhängigkeiten hinzugefügt, somit kann das Verfahren für das Hinzufügen von Attributen angewendet werden wofür bereits gezeigt wurde, dass AG' nur dann nicht zerlegbar ist, wenn dies bereits bei AG der Fall war.

Für den praktischen Aufbau von Mustern aus Basismustern – dem Ziel dieser Arbeit – ist es notwendig die dafür benötigte Komposition als zerlegungserhaltend zu zeigen:

Theorem 2. *Seien $AG \triangleq (G, A, R, B)$ eine zerlegbare Attributgrammatik mit abstrakter Syntax $G \triangleq (N, T, P, Z)$ und zwei Muster $\mathcal{M}_{G,1}$ und $\mathcal{M}_{G,2}$ gegeben mittels Musterattributierungstermen, dann ist die resultierende Attributgrammatik $AG'' \triangleq (\mathcal{M}_{G,2} \circ \mathcal{M}_{G,1})(AG)$ bestimmt als $\mathcal{M}_{G,2}(\mathcal{M}_{G,1}(AG))$ zerlegbar.*

Beweis. Angenommen AG'' wäre nicht zerlegbar, dann existieren folgende Fälle:

1. *die ursprüngliche Attributgrammatik AG ist nicht zerlegbar,*
2. *die nach Anwendung von $\mathcal{M}_{G,1}$ resultierende Attributgrammatik $AG' \triangleq \mathcal{M}_{G,1}(AG)$ ist nicht zerlegbar oder*
3. *AG'' ist nicht zerlegbar und die anderen Fälle sind nicht eingetreten.*

In Fall 1 ist bereits die Voraussetzung nicht erfüllt. In Fall 2 wird die Voraussetzung, dass $\mathcal{M}_{G,1}$ ein Muster ist verletzt und im letzten Fall, Fall 3, kann dies nur der Fall sein, wenn einer der anderen Fälle bereits eingetreten ist oder $\mathcal{M}_{G,2}$ kein Muster ist; dies steht im Widerspruch zu den Voraussetzungen.

Mit diesen beiden Thesen lässt sich zeigen dass Muster aufgebaut werden können. Die genaue Darstellung von Mustern unabhängig von einer Attributgrammatik nur bezüglich einer abstrakten Syntax bzw. unabhängig von der abstrakten Syntax sprengt den Rahmen dieser Arbeit. Bereits in [**?**,**?**,9] wurde eine frühe Form typischer Muster angewendet. Dabei wurde gezeigt, dass Laufzeit und Kompaktheitsgrad anderen Ansätzen, wie bspw. OCL in nichts nachstehen.

6 Zusammenfassung

In dieser Arbeit wurden typische Muster als Möglichkeit der knapperen Beschreibung der Sprachsemantik präsentiert. Es wurde gezeigt, dass und wie sich aus einer Reihe simpler Basismuster und deren Komposition neue Muster erstellen lassen. Ein wesentlicher Vorteil typischer Muster gegenüber üblichen Verfahren, wie bspw. in [29,38] oder [11] ist, dass die Berechenbarkeit garantiert ist. Werden typische Muster auf eine geordnete Attributgrammatik angewandt, ist sichergestellt, dass das Resultat mindestens zerlegbar ist. Eine zulässige Zerlegung bzw. eine lineare Anordnung, lässt sich ebenfalls ermitteln. Bereits Kastens hat in [27] gezeigt, dass jede zerlegbare Attributgrammatik durch Einführung zusätzlicher Abhängigkeiten in eine geordnete Attributgrammatik überführt werden kann.

Typische Muster überbrücken die große Lücke zwischen den grobkörnigen Modulansätzen wie diese in [25] und [7] vorgestellt werden und einfachen Erweiterungen für die [29,38,34] einen Überblick geben. Typische Muster können grundsätzlich wie entfernte Attribute oder Attribute höherer Ordnung oder inkrementeller Auswertung verwendet werden. Die Eigenschaften der in dieser Arbeit vorgestellten typischen Muster erlauben jedoch die Verwendung mit geordneten Attributgrammatiken, welche aufgrund ihrer Eigenschaften ein sehr günstiges Laufzeitverhalten zeigen und deswegen alternativen Ansätzen vorzuziehen sind[9,10].

Im Bereich der Definition der Sprachsemantik existieren eine Reihe weiterer Arbeiten, wie [19,15] bei der Aspekte bzw. Termersetzung genutzt werden um die Sprachsemantik erweiterbar zu gestalten. Eine Auf Termersetzung basierende Implementierung einer Sprache zur Definition der Namensanalyse wird hingegen in [33] vorgestellt. Das Ziel typischer Muster war jedoch nicht die Verwendung von Termersetzung – dies erzeugt üblicherweise langsamere Übersetzer – sondern die Definition eines Formalismus, der mit geordneten Attributgrammatiken einher geht.

Darüber hinaus sind typische Muster auch nicht mit den kompositionellen Ansätzen die bspw. in [14,16] vorgestellt werden zu vergleichen. Bei typischen Mustern ist die abstrakte Syntax der Attributgrammatik bspw. „fixiert“ und wird nicht durch Anwendung eines Musters (bzw. durch Anwendung der Komposition) geändert.

In weiteren Arbeiten muss untersucht werden wie diese Muster aus Basismustern aufgebaut werden müssen.

Literatur

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools (2Nd Edition). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2006)
2. Albreshne, A., Lahcen, A.A., Pasquier, J., Abdaladhem, A.: A framework and its associated process-oriented domain specific language for managing smart residential environments. International Journal of Smart Home 7(6), 377–392 (2013)
3. Albreshne, A., Lahcen, A.A.L., Pasquier, J.: Using a residential environment domain ontology for discovering and integrating smart objects in complex scenarios. Procedia Computer Science 32, 997 – 1002 (2014), http://www.sciencedirect.com/science/article/pii/S1877050914007248
4. Albreshne, A., Pasquier, J.: A domain specific language for high-level process control programming in smart buildings. Procedia Computer Science 63, 65 – 73 (2015), http://www.sciencedirect.com/science/article/pii/S1877050915024412
5. Backus, J.W., Bauer, F.L., Green, J., Katz, C., McCarthy, J., Naur, P., Perlis, A.J., Rutishauser, H., Samelson, K., Vauquois, B., Wegstein, J.H., van Wijngaarden, A., Woodger, M., van der Poel, W.L.: Revised report on the algorithmic language algol 60. Numerische Mathematik 4(1), 420–453 (1962), http://dx.doi.org/10.1007/BF01386340
6. Barendregt, H.P., et al.: The lambda calculus, vol. 2. North-Holland Amsterdam (1984)
7. Baum, B.: Another kind of modular attribute grammars. In: Kastens, U., Pfahler, P. (eds.) Compiler Construction, Lecture Notes in Computer Science, vol. 641, p. 44–50. Springer Berlin Heidelberg (1992), http://dx.doi.org/10.1007/3-540-55984-1_5
8. Bell, S., Gilbert, E.J.: Learning recursion with syntax diagrams. SIGCSE Bull. 6(3), 44–45 (Sep 1974), http://doi.acm.org/10.1145/988881.988890
9. Berg, C., Zimmermann, W.: Evaluierung von Möglichkeiten zur Implementierung von Semantischen Analysen für Domänenspezifische Sprachen (2014)
10. van Binsbergen, L.T., Bransen, J., Dijkstra, A.: Linearly ordered attribute grammars: With automatic augmenting dependency selection. In: Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation. pp. 49–60. PEPM '15, ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2678015.2682543
11. Boyland, J.T.: Remote attribute grammars. Journal of the ACM 52(4), 627–687 (Jul 2005), http://doi.acm.org/10.1145/1082036.1082042
12. Bransen, J., Middelkoop, A., Dijkstra, A., Swierstra, S.D.: The kennedy-warren algorithm revisited: Ordering attribute grammars. In: Russo, C., Zhou, N.F. (eds.) Practical Aspects of Declarative Languages: 14th International Symposium, PADL 2012, Philadelphia, PA, USA, January 23-24, 2012. Proceedings, pp. 183–197. Springer Berlin Heidelberg, Berlin, Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-27694-1_14
13. Campusano, M., Fabry, J.: Live robot programming: The language, its implementation, and robot {API} independence. Science of Computer Programming 133, Part 1, 1 – 19 (2017), http://www.sciencedirect.com/science/article/pii/S0167642316300697
14. Economopoulos, R., Fischer, B.: Higher-order transformations with nested concrete syntax. In: Proceedings of the Eleventh Workshop on Language Descriptions, Tools and Applications. p. 4. ACM (2011)

15. Ekman, T., Hedin, G.: Rewritable Reference Attributed Grammars. In: Odersky, M. (ed.) ECOOP 2004 – Object-Oriented Programming, Lecture Notes in Computer Science, vol. 3086, pp. 147–171. Springer Berlin Heidelberg (2004), http://dx.doi.org/10.1007/978-3-540-24851-4_7
16. Farrow, R., Marlowe, T.J., Yellin, D.M.: Composable attribute grammars: Support for modularity in translator design and implementation. In: Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 223–234. POPL '92, ACM, New York, NY, USA (1992), http://doi.acm.org/10.1145/143165.143210
17. Hedin, G.: Compiler Construction: 5th International Conference, CC '94 Edinburgh, U.K., April 7–9, 1994 Proceedings, chap. An overview of door attribute grammars, pp. 31–51. Springer Berlin Heidelberg, Berlin, Heidelberg (1994), http://dx.doi.org/10.1007/3-540-57877-3_3
18. Hedin, G.: Reference attributed grammars. Informatica (Slovenia) 24(3), 301–317 (2000)
19. Hedin, G., Magnusson, E.: Jastadd—an aspect-oriented compiler construction system. Science of Computer Programming 47(1), 37 – 58 (2003), http://www.sciencedirect.com/science/article/pii/S0167642302001090
20. Hess, S., Gross, A., Maier, A., Orfgen, M., Meixner, G.: Standardizing model-based in-vehicle infotainment development in the german automotive industry. In: Proceedings of the 4th International Conference on Automotive User Interfaces and Interactive Vehicular Applications. pp. 59–66. AutomotiveUI '12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2390256.2390265
21. Hindley, J.R., Seldin, J.P.: Lambda-calculus and combinators: an introduction, vol. 13. Cambridge University Press Cambridge (2008)
22. Jazayeri, M., Ogden, W.F., Rounds, W.C.: The intrinsically exponential complexity of the circularity problem for attribute grammars. Commun. ACM 18(12), 697–706 (Dec 1975), http://doi.acm.org/10.1145/361227.361231
23. Jazayeri, M., Walter, K.G.: Alternating semantic evaluator. In: Proceedings of the 1975 Annual Conference. pp. 230–234. ACM '75, ACM, New York, NY, USA (1975), http://doi.acm.org/10.1145/800181.810328
24. Kadhim, B., Waite, W.: Maptool — supporting modular syntax development. In: Gyimóthy, T. (ed.) Compiler Construction, Lecture Notes in Computer Science, vol. 1060, pp. 268–280. Springer Berlin Heidelberg (1996), http://dx.doi.org/10.1007/3-540-61053-7_67
25. Kastens, U., Waite, W.M.: Modularity and reusability in attribute grammars. Acta Informatica 31(7), 601–627 (1994), http://dx.doi.org/10.1007/BF01177548
26. Kastens, U., Waite, W.: An abstract data type for name analysis. Acta Informatica 28(6), 539–558 (1991), http://dx.doi.org/10.1007/BF01463944
27. Kastens, U.: Ordered attributed grammars. Acta Informatica 13(3), 229–256 (1980)
28. Kastens, U.: Liga: A language independent generator for attribute evaluators. Bericht der Reihe Informatik (63) (1989)
29. Kastens, U.: Attribute Grammars as a specification method. In: Alblas, H., Melichar, B. (eds.) Attribute Grammars, Applications and Systems, Lecture Notes in Computer Science, vol. 545, pp. 16–47. Springer Berlin Heidelberg (1991), http://dx.doi.org/10.1007/3-540-54572-7_2
30. Kastens, U., Hutt, B., Zimmermann, E.: GAG, a practical compiler generator, Lecture Notes in Computer Science, vol. 141. Springer-Verlag (1982)

31. Kennedy, K., Warren, S.K.: Automatic generation of efficient evaluators for attribute grammars. In: Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles on Programming Languages. pp. 32–49. POPL '76, ACM, New York, NY, USA (1976), http://doi.acm.org/10.1145/800168.811538
32. Knuth, D.E.: Semantics of context-free languages. Mathematical systems theory 2(2), 127–145 (1968), http://dx.doi.org/10.1007/BF01692511
33. Konat, G., Kats, L., Wachsmuth, G., Visser, E.: Software Language Engineering: 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers, chap. Declarative Name Binding and Scope Rules, pp. 311–331. Springer Berlin Heidelberg, Berlin, Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-36089-3_18
34. Koskimies, K.: Object-orientation in attribute grammars. In: Alblas, H., Melichar, B. (eds.) Attribute Grammars, Applications and Systems: International Summer School SAGA Prague, Czechoslovakia, June 4–13, 1991 Proceedings, pp. 297–329. Springer Berlin Heidelberg, Berlin, Heidelberg (1991), http://dx.doi.org/10.1007/3-540-54572-7_11
35. Lewis, P., Rosenkrantz, D., Stearns, R.: Attributed translations. Journal of Computer and System Sciences 9(3), 279 – 307 (1974), http://www.sciencedirect.com/science/article/pii/S0022000074800450
36. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. 37(4), 316–344 (Dec 2005), http://doi.acm.org/10.1145/1118890.1118892
37. Natori, S., Gondow, K., Imaizumi, T., Hagiwara, T., Katayama, T.: On eliminating type 3 circularities of ordered attribute grammars. In: Parigot, D., Mernik, M. (eds.) 2nd Int. Workshop on Attribute Grammars and their Applications (WAGA'99). pp. 93–112. INRIA Rocquencourt France (1999)
38. Paakki, J.: Attribute grammar paradigms – a high-level methodology in language implementation. ACM Comput. Surv. 27(2), 196–255 (Jun 1995), http://doi.acm.org/10.1145/210376.197409
39. Stewart, G., Gowda, M., Mainland, G., Radunovic, B., Vytiniotis, D., Agullo, C.L.: Ziria: A dsl for wireless systems programming. In: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 415–428. ASPLOS '15, ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2694344.2694368
40. Swierstra, D., Vogt, H.: Higher order attribute grammars. In: Alblas, H., Melichar, B. (eds.) Attribute Grammars, Applications and Systems: International Summer School SAGA Prague, Czechoslovakia, June 4–13, 1991 Proceedings, pp. 256–296. Springer Berlin Heidelberg, Berlin, Heidelberg (1991), http://dx.doi.org/10.1007/3-540-54572-7_10
41. Swierstra, S.D., Alcocer, P.R.A., Saraiva, J.: Designing and implementing combinator languages. In: International School on Advanced Functional Programming. pp. 150–206. Springer (1998)
42. Tienari, M.: On the definition of an attribute grammar. In: Jones, N.D. (ed.) Semantics-Directed Compiler Generation: Proceedings of a Workshop Aarhus, Denmark, January 1980, pp. 408–414. Springer Berlin Heidelberg, Berlin, Heidelberg (1980), http://dx.doi.org/10.1007/3-540-10250-7_31
43. Vogt, H.H., Swierstra, S.D., Kuiper, M.F.: Higher order attribute grammars. In: Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation. pp. 131–145. PLDI '89, ACM, New York, NY, USA (1989), http://doi.acm.org/10.1145/73141.74830

44. Waite, W.M., Goos, G.: Compiler construction. Springer-Verlag, New York (1984)
45. Wirth, N.: What can we do about the unnecessary diversity of notation for syntactic definitions? Commun. ACM 20(11), 822–823 (Nov 1977), http://doi.acm.org/10.1145/359863.359883

Structural type inference in Java-like languages

Martin Plümicke

Baden-Wuerttemberg Cooperative State University Stuttgart/Horb
Department of Computer Science
Florianstraße 15, D–72160 Horb
pl@dhbw.de

Abstract. In the past we considered type inference for Java with generics and lambdas. Our type inference algorithm determines nominal types in subjection to a given environment. This is a hard restriction as separate compilation of Java classes without relying on type informations of other classes is impossible. In this paper we present a type inference algorithm for a Java-like language, that infers structural types without a given environment. This allows separate compilation of Java classes without relying on type informations of other classes.

1 Introduction

In this paper we give an algorithm which is a generalization of an idea, that is given in [ADDZ05]. In the introducing example from [ADDZ05] the method `E m(B x){ return x.f1.f2; }` is given. The compilation algorithm generates the polymorphic typed Java expressions

$$\texttt{E m(B x)\{ return [[x:B].f1:}\alpha\texttt{].f2:}\beta\texttt{; \}},$$

where α and β are type variables. In this system m is applicable to instances of the class `B` with the field `f1` with the type α, where α must have a field `f2` with the type β and the constraint $\beta \leq^* \texttt{E}$. In this approach `B` and `E` are still nominal types.
We generalize this approach, such that also untyped methods like `m(x){ return x.f1.f2; }` can be compiled, that means the type of `x` and the return type are type variables, too.
Furthermore the results of our algorithms are Java-like programs, not byte-code, as in [ADDZ05], such that the typed programs are readable and the linking prcoess is reduced to a simple ckeck if a class implements a given interface.
This algorithm can be considered as generalization of our type inference algorithms [Plü15,Plü07].
Let us consider an example that shows the differences. For the following program no type can be inferred, as there is no type assumption for `elementAt`.

```
class A { m (v) { return v.elementAt(0); } }
```

Only with an import declaration `import java.util.Vector;` a type can be inferred.
The generalized algorithm which we give in this paper infers for `v` a structural type α, which has a method `elementAt`.

The basic idea

The result of our type inference algorithm is a parameterized class, where each inferred type is represented by a parameter that implements a new generated interfaces.

The paper is organized as follows. In the next section the language is introduced. In the third section we give the algorithm. Then we present a large example. Finally we close with a summary.

2 The language

We consider a core of a Java-like language without lambdas. In Figure 1 the syntax of the language is given. It is an extension of Featherweight Java [IPW01]. The syntax is differed between input and output syntax. The input is an untyped

Input syntax :

$\texttt{L} \quad ::= \texttt{class C extends (CT)}^* \ \{\overline{\texttt{f}};\ \overline{\texttt{M}}\}$

$\texttt{M} \quad ::= \texttt{m}(\overline{\texttt{x}})\{\ \texttt{return e};\ \}$

$\texttt{e} \quad ::= \texttt{x} \mid \texttt{e.f} \mid \texttt{e.m}(\overline{\texttt{e}}) \mid \texttt{new NCT}(\overline{\texttt{e}}) \mid \texttt{(CT)e}$

$\texttt{NCT} ::= \texttt{CT} \mid \texttt{C<}\overline{\texttt{TVar = T}}\texttt{>}$

$\texttt{T} \quad ::= \texttt{CT} \mid \texttt{TVar}$

$\texttt{CT} \quad ::= \texttt{C<}\overline{\texttt{CT}}\texttt{>}$

Output syntax :

$\texttt{L}_t \quad ::= \texttt{I}^* \ \texttt{CL}_t$

$\texttt{CL}_t \quad ::= \texttt{class C<}\overline{\texttt{TVar}}\texttt{>[}\overline{\texttt{CONS}}\texttt{] extends (CT)}^* \ \{\overline{\texttt{T f}};\overline{\texttt{M}_\texttt{t}}\}$

$\texttt{CONS} ::= \texttt{T extends T}$

$\texttt{MH} \quad ::= \texttt{T m}(\overline{\texttt{T}}\ \overline{\texttt{x}});$

$\texttt{M}_t \quad ::= \texttt{MH}\ \{\ \texttt{return e}_\texttt{t};\ \}$

$\texttt{e}_t \quad ::= \texttt{x : T} \mid \texttt{e.f : T} \mid \texttt{e.m}(\overline{\texttt{e}}) \texttt{ : T} \mid \texttt{new NCT}(\overline{\texttt{e}}) \texttt{ : CT} \mid \texttt{(CT)e : CT}$

$\texttt{I} \quad ::= \texttt{interface I<}\overline{\texttt{TVar}}\texttt{>}\{\overline{\texttt{T}}\ \overline{\texttt{f}};\overline{\texttt{MH}}\}$

Fig. 1. The syntax

Java program L and the output is a typed Java program $\texttt{L}_t$, including generated interfaces.
There are some extensions in comparison to usual Java. The class declarations in the output syntax have the form `class C<`$\overline{\texttt{TVar}}$`> [CONS]`. $\overline{\texttt{TVar}}$ are the generics

and `[CONS]` is a set of subtype constraints `T extends T'`, that must fulfill all instances of the class. In any class there is an implicit constructor with all fields (including them from the superclasses) as arguments. There is no differentiation between `extends` and `implements` declarations. Both are declared by `extends`. Interfaces can have fields. Furthermore, the use of the new-statement is allowed without assigning all generics. This is done by the syntax `C<TVar = CT>`. The not assigned generics are derived by the type inference algorithm.

3 The algorithm

The algorithm **TI** consists of three parts. First the function **TYPE** inserts types (usually type variables) to all sub-terms and collects constraints. Second, the function **construct** generates the interfaces and completes the constraints. Finally, the function **solve** unifies the type constraints and applies the unifier to the class.

In the following definition we give the different forms of constraints, that are collected in **TYPE**. This definition is oriented at [ADDZ05]:

Definition 1. (Type constraints)

- $c \lessdot c'$ *means* c has to be a subtype of c'.
- $\phi(\, c, f, c'\,)$ *means* c provides a field f with type c'.
- $\mu(\, c, m, \overline{c}, (c', \overline{c}')\,)$ *means* c provides a method m applicable to arguments of type $\overline{c}$, with return type c' and parameters of type $\overline{c}'$.

Note that $\mu(\, c, m, \overline{c}, (c', \overline{c}')\,)$ implicitly includes the constraints $\overline{c} \leq \overline{c}'$.

Let $<$ be the extends relation defined by the `Java` declarations und $\leq^*$ the corresponding subtyping relation.

The type–inference algorithm

Let `TypeAssumptions` be a set of assumptions, that can consists of assumptions for fields, methods and whole classes with fields and methods. The functions *fields* and *mtype* extracts the typed fields respectively the typed methods from a given class, as in [IPW01].

In the type inference algorithm we use the following name conventions for type variables:

$\delta_{\mathtt{A}}^{\mathtt{f}}$**:** Type variable for the field `f` in the class `A`.
$\alpha_{\mathtt{A}}^{\mathtt{m,i}}, \beta_{\mathtt{A}}^{\mathtt{m,i}}$**:** Type variable for the i-th argument of the method `m` in the class `A`.
$\overline{\alpha_{\mathtt{A}}^{\mathtt{m}}}, \overline{\beta_{\mathtt{A}}^{\mathtt{m}}}$**:** is an abbreviation for the tuple $\alpha_{\mathtt{A}}^{\mathtt{m,1}}, \ldots, \alpha_{\mathtt{A}}^{\mathtt{m,n}}$ respectively $\beta_{\mathtt{A}}^{\mathtt{m,1}}, \ldots, \beta_{\mathtt{A}}^{\mathtt{m,n}}$.
$\gamma_{\mathtt{A}}^{\mathtt{m}}$**:** Type variable for the return type of the method `m` in the class `A`.

The main function TI The main function **TI** calls the three functions **TYPE**, **construct**, and **solve**. The input is a set of type assumptions `TypeAssumptions` and an untyped Java class `L`. The result $\mathtt{L}_t$ is the typed Java class extended by a set of interfaces.

TI: $\mathtt{TypeAssumptions} \times \mathtt{L} \to \mathtt{L}_t$

TI $(\, Ass, \mathtt{class\ A\ extends\ \overline{B}\ \{\ \overline{f};\ \overline{M}\ \}}\,) =$
let
$(cl_t, C) = \mathbf{Type}(\, Ass, cl\,)$
$(I_1 \ldots I_m\ cl_t) = \mathbf{construct}(\, cl_t, C\,)$
in
$(I_1 \ldots I_m\ \mathbf{solve}(\, cl_t\,))$

The function TYPE The function **TYPE** inserts types (usually type variables) to all sub-terms and collects the constraints.

TYPE: $\mathtt{TypeAssumptions} \times \mathtt{L} \to \mathtt{L}_t \times \mathtt{ConstraintsSet}$

$\mathbf{TYPE}(\, Ass, \mathtt{class\ A\ extends\ \overline{B}\ \{\ \overline{f};\ \overline{M}\ \}}\,) =$ **let**
$fass := \{\, \mathtt{this.f} : \delta^{\mathtt{f}}_{\mathtt{A}} \mid \mathtt{f} \in \overline{\mathtt{f}}\,\} \cup \{\, \mathtt{this.f} : \mathtt{T} \mid \mathtt{T\ f} \in fields(\,\overline{\mathtt{B}}\,)\,\}$
$mass := \{\, \mathtt{this.m} : \overline{\alpha^{\mathtt{m}}_{\mathtt{A}}} \to \gamma^{\mathtt{m}}_{\mathtt{A}} \mid\ \mathtt{m}(\overline{\mathtt{x}})\{\ \mathtt{return\ e};\ \} \in \overline{\mathtt{M}}\,\} \cup$
$\{\, \mathtt{this.m} : \overline{aty} \to rty \mid\ mtype(\,\mathtt{m}, \overline{\mathtt{B}}\,) = \overline{aty} \to rty\,\}$
$AssAll = Ass \cup fass \cup mass \cup \{\, \mathtt{this} : \mathtt{A}\,\}$
For $\mathtt{m}(\overline{\mathtt{x}})\{\ \mathtt{return\ e};\ \} \in \overline{\mathtt{M}}$ {
$Ass = AssAll \cup \{\, \mathtt{x}_j : \alpha^{\mathtt{m,j}}_{\mathtt{A}} \mid \overline{\mathtt{x}} = \mathtt{x}_1 \ldots \mathtt{x}_n\,\}$
$(\mathtt{e}_t : rty, C') = \mathbf{TYPEExpr}(\, Ass, \mathtt{e}\,)$
$C = (C \cup C')[\gamma^{\mathtt{m}}_{\mathtt{A}} \mapsto rty]$}
$\overline{\mathtt{M}}_t = \{\, rty\ \ \mathtt{m}(\overline{\alpha^{\mathtt{m}}_{\mathtt{A}}\ \mathtt{x}})\{\ \mathtt{return\ e_t};\ \} \mid \mathtt{m}(\overline{\mathtt{x}})\{\ \mathtt{return\ e};\ \} \in \overline{\mathtt{M}}\,\}$
in$(\mathtt{class\ A\ extends\ \overline{B}\ \{\ \overline{\delta^{\mathtt{f}}_{\mathtt{A}}\mathtt{f}}; \overline{\mathtt{M}_t}\ \}}, C)$

The function **TYPEExpr** inserts types into the expressions and collects the corresponding constraints. It is given for all cases of expressions `e`.

TYPEExpr: $\mathtt{TypeAssumptions} \times \mathtt{e} \to \mathtt{e}_t \times \mathtt{ConstraintsSet}$

$\mathbf{TYPEExpr}(\, Ass, \mathtt{x}\,) =$ **let** $(\mathtt{x} : \theta) \in Ass$ **in** $(\mathtt{x} : \theta, \emptyset)$

TYPEExpr for field-application: First, the type of the receiver is determined. Then it is differed between fields with and without known receiver types. In the known case the types are introduced. Otherwise a constraint is generated that demands a corresponding field in the type.

$\mathbf{TYPEExpr}(\, Ass, \mathtt{e.f}\,) =$
let
$\underline{(\mathtt{e}_t : ty, C)} = \mathbf{TYPEExpr}(\, Ass, \mathtt{e}\,)$
in
if $(ty$ *is no type variable* $) \,\&\&\, (ty \in Ass) \,\&\&(rty\ \mathtt{f} \in \mathit{fields}(\, ty\,))$
then $(((\mathtt{e}_t : ty).\mathtt{f}) : rty, C)$
else $(((\mathtt{e}_t : ty).\mathtt{f}) : \delta^{\mathtt{f}}_{\mathtt{ty}}, \{\, \phi(\, ty, \mathtt{f}, \delta^{\mathtt{f}}_{\mathtt{ty}}\,)\,\} \cup C)$

TYPEExpr for method-call: First, the types of the receiver and the arguments are determined, recursively. Then it is differed between methods with and without known receiver types. In the known case a subtype relation is introduced. Otherwise a constraint is generated that demands a corresponding method in the type.

$\mathbf{TYPEExpr}(\, Ass, \mathtt{e_0.m}(\overline{e})\,) = \mathbf{let}$
$\quad (\mathtt{e_{0_t}} : ty_0, C_0) = \mathbf{TYPEExpr}(\, Ass, \mathtt{e_0}\,)$
$\quad (\mathtt{e_{i_t}} : ty_i, C_i) = \mathbf{TYPEExpr}(\, Ass, \mathtt{e_i}\,), \forall 1 \leqslant i \leqslant n$
$\mathbf{in}$
$\quad \mathbf{if}\ (ty_0\ \textit{is no type variable})\ \&\&\ (ty_0 \in Ass)\ \&\&\ (\mathit{mtype}(\mathtt{m}, ty_0\,) = \overline{aty} \to rty)$
$\quad \mathbf{then}$
$\qquad ((\mathtt{e_{0_t}} : ty_0).\mathtt{m}(\mathtt{e_{1_t}} : ty_1, \ldots, \mathtt{e_{n_t}} : ty_n) : rty, (C_0 \cup \bigcup_i C_i) \cup \{\, \overline{ty \lessdot aty}\,\})$
$\quad \mathbf{else}$
$\qquad ((\mathtt{e_{0_t}} : ty_0).\mathtt{m}(\mathtt{e_{1_t}} : ty_1, \ldots, \mathtt{e_{n_t}} : ty_n) : \gamma^{\mathtt{m}}_{\mathtt{ty_0}},$
$\qquad (C_0 \cup \bigcup_i C_i) \cup \{\, \mu(\, ty_0, \mathtt{m}, \overline{ty}, (\gamma^{\mathtt{m}}_{\mathtt{ty_0}}, \overline{\beta^{\mathtt{m}}_{\mathtt{ty_0}}})\,)\,\})$
$\qquad$ where $\overline{\beta^{\mathtt{m}}_{\mathtt{ty_0}}}$ and $\gamma^{\mathtt{m}}_{\mathtt{ty_0}}$ are fresh type variables.

TYPEExpr for the new-statement: The use of the new-statement is allowed without assigning all generics. This is done by the syntax `C<`$\overline{\texttt{TVar = CT}}$`>`. First, fresh type variables are introduced in the assumptions of the corresponding class. Then the types of the arguments are determined. Finally, the assigned generics are introduced and the subtype relations between the argument types and the fields of the class and its super classes are added.

$\mathbf{TYPEExpr}(\, Ass \cup \{\, \mathtt{class\ A<}\overline{\mathtt{T}}\mathtt{>[}C_\mathtt{A}\mathtt{]\ extends\ }\overline{\mathtt{B}}\ \{\overline{\mathtt{T_A\ f;}}\ \overline{\mathtt{M_t}}\}\,\}, \mathtt{new\ A<}S\mathtt{>}(\,\overline{e}\,)\,) =$
$\quad$ where $S = [\mathtt{T}_{\pi(1)} = \tau_1, \ldots, \mathtt{T}_{\pi(k)} = \tau_k]$ with $k \leq n$ for $|\overline{\mathtt{T}}| = n$
$\quad \mathbf{let}$
$\qquad \overline{\nu}$ fresh type variables, that substitute $\overline{\mathtt{T}}$ and all type variables of C_A
$\qquad$ in class $\mathtt{A}$
$\qquad S' = S[\overline{\mathtt{T} \mapsto \nu}]$
$\qquad (\mathtt{e_{i_t}} : ty_i, C_i) = \mathbf{TYPEExpr}(\, Ass, e_i\,), \forall 1 \leqslant i \leqslant m$
$\quad \mathbf{in}$
$\qquad (\mathtt{new\ A<}S'\mathtt{>}(\,\mathtt{e_{1_t}} : \mathtt{ty_1}, \ldots, \mathtt{e_{m_t}} : \mathtt{ty_m}\,) : \mathtt{A<}\overline{\nu}[\nu \mapsto \tau \mid \nu = \tau \in S']\mathtt{>},$
$\qquad (\bigcup_i C_i) \cup C_\mathtt{A}[\overline{\nu \mapsto \tau}] \cup \{\, \overline{ty \lessdot \mathtt{T_B}\ \mathtt{T_A}[\overline{\nu \mapsto \tau}]}\,\}$
$\qquad$ where $\mathit{fields}(\,\overline{\mathtt{B}}\,) = \overline{\mathtt{T_B\ g}}$

$\mathbf{TYPEExpr}(\, Ass, (\mathtt{A})\mathtt{e}\,) =$
$\quad \mathbf{let}$
$\qquad (ty, C) = \mathbf{TYPEExpr}(\, Ass, \mathtt{e}\,)$
$\quad \mathbf{in}$
$\qquad ((\mathtt{A})\mathtt{e} : \mathtt{A}, C)$

The function construct The function **construct** takes the result from **TYPE**, a typed class and a set of constraints. It generates for any type `ty1`, `ty2` occuring

in constraints $\phi(\, ty1, \mathtt{f}, \delta\,)$ or $\mu(\, ty2, \mathtt{m}, \overline{\alpha}, (\gamma, \overline{\beta})\,)$ corresponding interfaces with the demanded fields and methods.

construct : $\mathtt{L}_t \times \mathtt{ConstraintsSet} \to \mathtt{L}_t$

construct(`class A extends` $\overline{\mathtt{B}}$ $\{\, \overline{\delta_{\mathtt{A}}\mathtt{f};\ \mathtt{M_t}}\,\}, C\,)$ {
  $C_{\mathtt{A}} = \{\, \alpha \lessdot \beta \mid \alpha \lessdot \beta \in C\,\} \cup \{\, \overline{\alpha \lessdot \beta} \mid \mu(\, ty, \mathtt{m}, \overline{\alpha}, (\gamma, \overline{\beta})\,) \in C\,\}$
  if C_A contains a constraint $ty \lessdot ty'$, where ty and ty'
    are not type variables and $ty \not\leq^* ty'$ **then fail exit**
  new_interf $= \{\, \iota \mid \phi(\, \iota, \mathtt{f}, \gamma\,) \in C\,\} \cup \{\, \iota \mid \mu(\, \iota, \mathtt{m}, \overline{\beta}, (\gamma, \overline{\beta'})\,) \in C\,\}$
  For $\iota \in$ new_interf {
    $I_\iota =$ `interface` ι `< > { }` is generated
    inh_tyterm = "ι < >"
    For `f` with $\phi(\, \iota, \mathtt{f}, \delta\,) \in C$ {
      An field `f` and a fresh type variable T is added to the interface ι:

      $I_\iota =$ `interface` ι `<`*args*`, T> {... T f; ... }`

      `T` is instantiated by δ in inh_tyterm: inh_tyterm = "ι<*args'*,δ>" }
    For `m` with $\mu(\, \iota, \mathtt{m}, \overline{\beta}, (\gamma, \beta'_1, \ldots, \beta'_n)\,) \in C$ {
      A method signature for `m` and fresh type variable `T`, $\overline{\mathtt{T}}$
      are introduced into the interface ι:

      $I_\iota =$ `interface` ι `<`*args*`,T,`$\mathtt{T}_1$`, ..., `$\mathtt{T}_n$`> {`
        fields
        meth_sigs
        `T m(`$\mathtt{T}_1$`, ..., `$\mathtt{T}_n$`);`
      `}`

      `T` and $\overline{\mathtt{T}}$ are instantiated by γ and $\overline{\beta'}$ in inh_tyterm:
        inh_tyterm = "ι<*args'*,$\gamma, \overline{\beta'}$>" }
    $C_{\mathtt{A}} = C_{\mathtt{A}}[\iota \mapsto X] \cup \{\, X \lessdot \mathit{inh\_tyterm}\,\}$, where X is a fresh type variable
    $\sigma = \sigma \cup \{\, \iota \mapsto X\,\}$ }
  tv = **typevar**$(\, \overline{\sigma(\, \delta^{\mathtt{A}}_{\mathtt{f}}\,)}\,) \cup$ **typevar**$(\, \overline{\sigma(\, \mathtt{M_t}\,)}\,)$
  return $\overline{I_\iota}$ `class A<tv>[`$C_{\mathtt{A}}$`] extends` $\overline{\mathtt{B}}$ `{` $\overline{\sigma(\, \delta^{\mathtt{A}}_{\mathtt{f}}\,)\ \mathtt{f};}$ $\overline{\sigma(\, \mathtt{M_t}\,)}$`}`

The function solve The function **solve** takes the result of **construct** and solves the constraints of the class by a type unification algorithm, such that the constraints contains only pairs with at least one type variable.

First we consider the type unification. In [Plü09] we gave a type unification for Java 5.0 types. This algorithm is finitary but not unitary, as pairs $\mathtt{T} \lessdot ty$ are solved by substituting `T` by all subtypes of ty. Now we give a different version of the algorithm where $\mathtt{T} \lessdot ty$ is not solved.

$$\text{(reduce1)}\ \frac{C \cup \{\, C\langle\theta_1,\ldots,\theta_n\rangle \lessdot D\langle\theta'_1,\ldots,\theta'_n\rangle \,\}}{C \cup \{\, \theta_1 \doteq \theta'_1,\ldots,\theta_n \doteq \theta'_n \,\}}$$
where $C\langle T_1,\ldots,T_n\rangle \leq^* D\langle T_1,\ldots,T_n\rangle$ with T_i are type variables

$$\text{(reduce2)}\ \frac{C \cup \{\, C\langle\theta_1,\ldots,\theta_n\rangle \doteq C\langle\theta'_1,\ldots,\theta'_n\rangle \,\}}{C \cup \{\, \theta_1 \doteq \theta'_1,\ldots,\theta_n \doteq \theta'_n \,\}}$$

$$\text{(adapt1)}\ \frac{C \cup \{\, D\langle\theta_1,\ldots,\theta_n\rangle \lessdot D'\langle\theta'_1,\ldots,\theta'_m\rangle \,\}}{C \cup \{\, D'\langle\tilde{\theta}'_1,\ldots,\tilde{\theta}'_m\rangle[T_i \mapsto \theta_i \mid 1 \leqslant i \leqslant n] \doteq D'\langle\theta'_1,\ldots,\theta'_m\rangle \,\}}$$
where $(D\langle T_1,\ldots,T_n\rangle \leq^* D'\langle\tilde{\theta}'_1,\ldots,\tilde{\theta}'_m\rangle)$ with T_i are type variables

$$\text{(adapt2)}\ \frac{C \cup \{\, D\langle\theta_1,\ldots,\theta_n\rangle \lessdot S_1, S_1 \lessdot S_2,\ldots,S_{k-1} \lessdot S_k, S_k \lessdot D'\langle\theta'_1,\ldots,\theta'_m\rangle \,\}}{C \cup \{\, \sigma(\, D\langle\theta_1,\ldots,\theta_n\rangle\,) \lessdot S_1, S_1 \lessdot S_2,\ldots, S_{k-1} \lessdot S_k, S_k \lessdot \sigma(\, D'\langle\theta'_1,\ldots,\theta'_m\rangle\,) \cup \sigma \,\}}$$
where
- $k \geq 1$
- $S_i \in TV$ and
- $(D\langle\theta_1,\ldots,\theta_n\rangle \not\leq^* D'\langle\theta'_1,\ldots,\theta'_m\rangle)$ but $(D\langle T_1,\ldots,T_n\rangle \leq^* D'\langle\tilde{\theta}'_1,\ldots,\tilde{\theta}'_m\rangle)$ with $T_i \in TV$
- $\sigma = \mathbf{Unify}^1(\,\{\, D'\langle\tilde{\theta}'_1,\ldots,\tilde{\theta}'_m\rangle[T_i \mapsto \theta_i \mid 1 \leqslant i \leqslant n] \doteq D'\langle\theta'_1,\ldots,\theta'_m\rangle \,\}\,)$

$$\text{(erase1)}\ \frac{C \cup \{\, \theta \lessdot \theta' \,\}}{C}\ \theta \leq^* \theta'$$

$$\text{(erase2)}\ \frac{C \cup \{\, \theta \doteq \theta' \,\}}{C}\ \theta = \theta'$$

$$\text{(swap)}\ \frac{C \cup \{\, \theta \doteq T \,\}}{C \cup \{\, T \doteq \theta \,\}}\ \theta \notin TV,\ T \in TV$$

$$\text{(subst)}\ \frac{C \cup \{\, T \doteq \theta \,\}}{C[T \mapsto \theta] \cup \{\, T \doteq \theta \,\}}\ T \in TV \text{ and } T \text{ occurs in } C \text{ but not in } \theta$$

$$\text{(refl)}\ \frac{C \cup \{\, \theta \lessdot T_1, T_1 \lessdot T_2,\ldots,T_{n-1} \lessdot T_n, T_n \lessdot \theta \,\}}{C \cup \{\, T_i = \theta \mid 1 \leqslant i \leqslant n \,\}}$$

Fig. 2. Java type unification

The algorithm **TUnify**(C) is given by the rules (Figure 2) application the most often as possible. If C contains finally of pairs $\mathtt{T} \doteq ty$, $\mathtt{T} \lessdot ty$, or $ty \lessdot \mathtt{T}$ then C is the result, otherwise the algorithm fails.

Lemma 1 (Termination). The algorithm **TUnify** terminates.

Lemma 2 (Soundness of TUnify). If a substitution σ is a solution of a constraint set C then σ is also a solution of **TUnify**(C).

Lemma 3 (Completeness). Let C be a set of constraints C and σ' a solution. For $\sigma = \{\, \mathtt{T} \mapsto ty \mid \mathtt{T} \doteq ty \in \mathbf{TUnify}(\, C\,)\,\}$ there are substitutions σ'' and σ_{rest}, such that $\sigma' = \sigma'' \circ ((\sigma_{rest} \circ \sigma) \cup \sigma_{rest})$.

Remark 1 (Most general unifier of TUnify). The substitution σ_{rest} is a solution of the remaining pairs $\mathtt{T} \lessdot ty$ and $ty \lessdot \mathtt{T}$. For any solution σ' there is substitution σ'_{rest}, such that $\sigma'_{rest} \circ \sigma$ is a most general unifier.

In the following we prove the two lemmata of *soundness* and *completeness*.

Proof. We do the prove by showing soundness and completeness for all type unification rules. We do this as we show that the solutions before and after application are the same.

reduce1: From the type term construction follows, if and only if σ is a solution of $\{\, C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>} \lessdot D\texttt{<}\theta'_1, \ldots, \theta'_n\texttt{>}\,\}$ it is also a solution of $\{\, \theta_1 \doteq \theta'_1, \ldots, \theta_n \doteq \theta'_n\,\}$, if $C\texttt{<}T_1, \ldots, T_n\texttt{>} \leq^* D\texttt{<}T_1, \ldots, T_n\texttt{>}$.

adapt1: From the type term construction follows iff

$$D\texttt{<}T_1, \ldots, T_n\texttt{>} \leq^* D'\texttt{<}\tilde{\theta}'_1, \ldots, \tilde{\theta}'_m\texttt{>}, \text{ where } T_i \text{ are type variables,}$$

then

$$D\texttt{<}T_1, \ldots, T_n\texttt{>}[T_i \mapsto \theta_i \mid 1 \leqslant i \leqslant n] \leq^* D'\texttt{<}\tilde{\theta}'_1, \ldots, \tilde{\theta}'_m\texttt{>}[T_i \mapsto \theta_i \mid 1 \leqslant i \leqslant n].$$

This means iff σ is a solution of $\{\, D\texttt{<}\theta_1, \ldots, \theta_n\texttt{>} \lessdot D'\texttt{<}\theta'_1, \ldots, \theta'_m\texttt{>}\,\}$ it is also a solution of $\{\, D'\texttt{<}\tilde{\theta}'_1, \ldots, \tilde{\theta}'_m\texttt{>}[T_i \mapsto \theta_i \mid 1 \leqslant i \leqslant n] \doteq D'\texttt{<}\theta'_1, \ldots, \theta'_m\texttt{>}\,\}$

adapt2: As $\leq^*$ is a partial ordering and a partial ordering is transitiv from the soundness and completeness **adapt1** follows the soundness und completeness of **adapt2**.

refl: As $\leq^*$ is a partial ordering and a partial ordering is reflexive soundness and completeness follows directly.

erase1: obvious.

reduce2, erase2, swap, subst: These rule corresponds to the rules in [MM82]. Therefore soundness and completeness is given.

Now we give the function **solve**, where **TUnify** is called.

solve: $\mathtt{L}_t \to \mathtt{L}_t$

solve(`class A<`$\overline{\mathtt{T}}$`>[`$C_{\mathtt{A}}$`] extends` $\overline{\mathtt{B}}$ `{`$\overline{\texttt{ty f}}$`;` $\overline{\mathtt{M_t}}$ `}`) =
  let
    $subst = \mathbf{TUnify}(\, C_{\mathtt{A}}\,)$
    $\sigma = \{\, \mathtt{T} \mapsto ty \mid \mathtt{T} \doteq ty \in subst\,\}$
    $cs = \{\, \mathtt{T} \lessdot ty \mid \mathtt{T} \lessdot ty \in subst\,\} \cup \{\, ty \lessdot \mathtt{T} \mid ty \lessdot \mathtt{T} \in subst\,\}$
    $\overline{\mathtt{T}}_{new} = \bigcup_{[(\texttt{rty m(}\overline{\texttt{aty x}}\texttt{)\{ return e; \}})\in\overline{\sigma(\mathtt{M_t})}]}(\mathsf{TVar}(\,\texttt{rty}\,) \cup \mathsf{TVar}(\,\texttt{aty}\,))$
  in
    if is_solvable(cs) `then`
      `class A<`$\overline{\mathtt{T}}_{new}$`>[`$cs$`] extends` $\overline{\mathtt{B}}$ `{`$\overline{\sigma(\,\texttt{ty}\,)\ \texttt{f}}$`;` $\overline{\sigma(\,\mathtt{M_t}\,)}$ `}`
    else *fail*

4 Example

In this section we give an example, that shows first a structural typing of a class independent from any environment. Then a concrete implementation of this class is given.

Example 1. Let the following class be given

```
class A {
    mt(x, y, z) { return x.sub(y).add(z); }
}
```

First **TYPE** is applied:

$$
\begin{array}{l}
\textbf{TYPE}(\,\emptyset, \mathtt{A}\,) \\
\quad \underline{\mathit{mass}} = \{\, \mathtt{this.mt} : (\alpha_{\mathtt{A}}^{\mathtt{mt},1}, \alpha_{\mathtt{A}}^{\mathtt{mt},2}, \alpha_{\mathtt{A}}^{\mathtt{mt},3}) \to \gamma_{\mathtt{A}}^{\mathtt{mt}} \,\} \\
\quad \underline{\mathit{AssAll}} = \mathit{mass} \cup \{\, \mathtt{this} : \mathtt{A} \,\} \\
\quad \underline{\mathtt{mt} \in \overline{M}}: \\
\qquad \underline{\mathit{Ass}} = \mathit{AssAll} \cup \{\, \mathtt{x} : \alpha_{\mathtt{A}}^{\mathtt{mt},1}, \mathtt{y} : \alpha_{\mathtt{A}}^{\mathtt{mt},2}, \mathtt{z} : \alpha_{\mathtt{A}}^{\mathtt{mt},3} \,\} \\
\qquad \textbf{TYPEExpr}(\,\mathit{Ass}, \mathtt{x.sub(y).add(z)}\,) \\
\qquad\quad \textbf{TYPEExpr}(\,\mathit{Ass}, \mathtt{x.sub(y)}\,) \\
\qquad\qquad \textbf{TYPEExpr}(\,\mathit{Ass}, \mathtt{x}\,) \\
\qquad\qquad\quad \textbf{Result}\ (\mathtt{x} : \alpha_{\mathtt{A}}^{\mathtt{mt},1}, \emptyset) \\
\qquad\qquad \textbf{TYPEExpr}(\,\mathit{Ass}, \mathtt{y}\,) \\
\qquad\qquad\quad \textbf{Result}\ (\mathtt{y} : \alpha_{\mathtt{A}}^{\mathtt{mt},2}, \emptyset) \\
\qquad\qquad C = \{\, \mu(\, \alpha_{\mathtt{A}}^{\mathtt{mt},1}, \mathtt{sub}, \alpha_{\mathtt{A}}^{\mathtt{mt},2}, (\gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}, \beta^{\mathtt{sub},1}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}})\,)\,\} \\
\qquad\qquad \textbf{Result}\ ([[\mathtt{x} : \alpha_{\mathtt{A}}^{\mathtt{mt},1}].\mathtt{sub}([\mathtt{y} : \alpha_{\mathtt{A}}^{\mathtt{mt},2}]) : \gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}], C) \\
\qquad\quad \textbf{TYPEExpr}(\,\mathit{Ass}, \mathtt{z}\,) \\
\qquad\qquad \textbf{Result}\ (\mathtt{z} : \alpha_{\mathtt{A}}^{\mathtt{mt},3}, \emptyset) \\
\qquad\quad C = C \cup \{\, \mu(\, \gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}, \mathtt{add}, \alpha_{\mathtt{A}}^{\mathtt{mt},3}, (\gamma^{\mathtt{add}}_{\gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}}, \beta^{\mathtt{add},1}_{\gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}}})\,)\,\} \\
\qquad\quad \textbf{Result}\ ([[[\mathtt{x} : \alpha_{\mathtt{A}}^{\mathtt{mt},1}].\mathtt{sub}([\mathtt{y} : \alpha_{\mathtt{A}}^{\mathtt{mt},2}]) : \gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}].\mathtt{add}(\mathtt{z} : \alpha_{\mathtt{A}}^{\mathtt{mt},3}) : \gamma^{\mathtt{add}}_{\gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}}], \\
\qquad\qquad\qquad C) =: \mathtt{e_t} \\
\textbf{Result}(cl_t, C) \\
\text{with}\quad cl_t := \mathtt{class\ A\ \{} \\
\qquad\qquad\qquad \gamma^{\mathtt{add}}_{\gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}}\ \mathtt{mt}(\alpha_{\mathtt{A}}^{\mathtt{mt},1}\ \mathtt{x},\ \alpha_{\mathtt{A}}^{\mathtt{mt},2}\ \mathtt{y},\ \alpha_{\mathtt{A}}^{\mathtt{mt},3}\ \mathtt{z})\ \mathtt{\{\ return\ e_t;\ \}} \\
\qquad\qquad \mathtt{\}} \\
\text{and}\quad C = \{\, \mu(\, \alpha_{\mathtt{A}}^{\mathtt{mt},1}, \mathtt{sub}, \alpha_{\mathtt{A}}^{\mathtt{mt},2}, (\gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}, \beta^{\mathtt{sub},1}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}})\,), \\
\qquad\qquad\quad \mu(\, \gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}, \mathtt{add}, \alpha_{\mathtt{A}}^{\mathtt{mt},3}, (\gamma^{\mathtt{add}}_{\gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}}, \beta^{\mathtt{add},1}_{\gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}}})\,)\,\}
\end{array}
$$

Second, **construct** is applied to **TYPE**'s result:

construct$(\, cl_t, C\,)$:

$C_{\mathtt{A}} = \{\, \alpha_{\mathtt{A}}^{\mathtt{mt},2} \lessdot \beta^{\mathtt{sub},1}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}},\quad \alpha_{\mathtt{A}}^{\mathtt{mt},3} \lessdot \beta^{\mathtt{add},1}_{\gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}} \,\}$

new_interf $= \{\, \alpha_{\mathtt{A}}^{\mathtt{mt},1}, \gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}} \,\}$

$\underline{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}$**:** For $\mu(\, \alpha_{\mathtt{A}}^{\mathtt{mt},1}, \mathtt{sub}, \alpha_{\mathtt{A}}^{\mathtt{mt},2}, (\gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}, \beta^{\mathtt{sub},1}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}})\,) \in C$: the following interface is constructed:

```
interface α^mt,1_A <T, T1> { T sub(T1 x); }
```

inh_tyterm $= \alpha_{\mathtt{A}}^{\mathtt{mt},1}\mathtt{<}\gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}, \beta^{\mathtt{sub},1}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}\mathtt{>}$

$C_{\mathtt{A}} = C_{\mathtt{A}} \cup \{\, \mathtt{X1} \lessdot \text{inh\_tyterm} \,\}$

$\sigma = \{\, \alpha_{\mathtt{A}}^{\mathtt{mt},1} \mapsto \mathtt{X1} \,\}$

$\underline{\gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}}$**:** For $\mu(\, \gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}, \mathtt{add}, \alpha_{\mathtt{A}}^{\mathtt{mt},3}, (\gamma^{\mathtt{add}}_{\gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}}, \beta^{\mathtt{add},1}_{\gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}})\,) \in C$ the following interface is constructed:

```
interface γ^sub_α^mt,1_A<T, T1> { T add(T1 x); }
```

inh_tyterm $= \gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}\mathtt{<}\gamma^{\mathtt{add}}_{\gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}}, \beta^{\mathtt{add},1}_{\gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}}\mathtt{>}$

$C_{\mathtt{A}} = C_{\mathtt{A}} \cup \{\, \mathtt{X2} \lessdot \text{inh\_tyterm} \,\}$

$\sigma = \sigma \cup \{\, \gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}} \mapsto \mathtt{X2} \,\}$

tv $= \{\, \mathtt{X1}, \alpha_{\mathtt{A}}^{\mathtt{mt},2}, \alpha_{\mathtt{A}}^{\mathtt{mt},3}, \gamma^{\mathtt{add}}_{\gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}} \,\}$

It holds $\sigma = \{\alpha_{\mathtt{A}}^{\mathtt{mt},1} \mapsto \mathtt{X1},\ \gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}} \mapsto \mathtt{X2}\,\}$

and $C_{\mathtt{A}} = \{\, \alpha_{\mathtt{A}}^{\mathtt{mt},2} \lessdot \beta^{\mathtt{sub},1}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}},\ \alpha_{\mathtt{A}}^{\mathtt{mt},3} \lessdot \beta^{\mathtt{add},1}_{\gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}},\ \mathtt{X1} \lessdot \alpha_{\mathtt{A}}^{\mathtt{mt},1}\mathtt{<X2}, \beta^{\mathtt{sub},1}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}\mathtt{>},\ \mathtt{X2} \lessdot \gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}\mathtt{<}\gamma^{\mathtt{add}}_{\gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}}, \beta^{\mathtt{add},1}_{\gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}}\mathtt{>}\,\}$

The result class is given as:

```
class A <X1, α^mt,2_A, α^mt,3_A, γ^add_γ^sub_α^mt,1_A>[
    α^mt,2_A extends β^sub,1_α^mt,1_A
    α^mt,3_A extends β^add,1_γ^sub_α^mt,1_A ,
    X1 extends α^mt,1_A<X2, β^sub,1_α^mt,1_A>,
```

```
    X2 extends γ^{sub}_{α^{mt,1}_A}<γ^{add}_{γ^{sub}_{α^{mt,1}_A}}, β^{add,1}_{γ^{sub}_{α^{mt,1}_A}}>
]
{
    γ^{add}_{γ^{sub}_{α^{mt,1}_A}} mt(X1 x, α^{mt,2}_A y, α^{mt,3}_A z) { return x.sub(y).add(z); }
}
```

As

$$C_{\mathtt{A}} = \{\, [\alpha_{\mathtt{A}}^{\mathtt{mt},2} \lessdot \beta^{\mathtt{sub},1}_{\alpha^{\mathtt{mt},1}_{\mathtt{A}}}],\ [\alpha_{\mathtt{A}}^{\mathtt{mt},3} \lessdot \beta^{\mathtt{add},1}_{\gamma^{\mathtt{sub}}_{\alpha^{\mathtt{mt},1}_{\mathtt{A}}}}],\ [\mathtt{X1} \lessdot \alpha_{\mathtt{A}}^{\mathtt{mt},1}\mathtt{<X2},\beta^{\mathtt{sub},1}_{\alpha^{\mathtt{mt},1}_{\mathtt{A}}}\mathtt{>}],$$
$$[\mathtt{X2} \lessdot \gamma^{\mathtt{sub}}_{\alpha^{\mathtt{mt},1}_{\mathtt{A}}}\mathtt{<}\gamma^{\mathtt{add}}_{\gamma^{\mathtt{sub}}_{\alpha^{\mathtt{mt},1}_{\mathtt{A}}}},\beta^{\mathtt{add},1}_{\gamma^{\mathtt{sub}}_{\alpha^{\mathtt{mt},1}_{\mathtt{A}}}}\mathtt{>}] \,\}$$

is in solved form, **TUnify** respectively **solve** changes nothing. This means the result of **construct** is the result of **TI**.

In the following we extend the example, such that an instance of class `A` is used. For this implementations of the inerfaces $\alpha_{\mathtt{A}}^{\mathtt{mt},1}$ and $\gamma^{\mathtt{sub}}_{\alpha^{\mathtt{mt},1}_{\mathtt{A}}}$ must be given. We give one class `myInteger`, which implements both interfaces:

```
class myInteger extends α^{mt,1}_A<myInteger, myInteger>,
                        γ^{sub}_{α^{mt,1}_A}<myInteger, myInteger> {
    Integer i;
    myInteger sub(myInteger x) { return new myInteger(i - x.i); }
    myInteger add(myInteger x) { return new myInteger(i + x.i); } }
```

In the class `Main` an instance of `A` is used and the method `mt` is called.

```
class Main {
    main() { return new A<>()
                    .mt(new myInteger(2),
                        new myInteger(1),
                        new myInteger(3)); }
}
```

We call **TI** for `Main` with the set of assumptions Ass consisting of the class `A` and the class `myInteger`.

In **TYPEExpr**(Ass, `new A<>()`) the class `A` gets fresh type variables:

```
class A <ν1,ν3,ν4,ν6>
        [ν3 extends ν5,
         ν4 extends ν7,
         ν1 extends α^{mt,1}_A<ν2,ν5>,
         ν2 extends γ^{sub}_{α^{mt,1}_A}<ν6,ν7>]{
     ν6 mt(ν1 x, ν3 y, ν4 z) { return x.sub(y).add(z); }
}
```

The result of **TYPEExpr**(Ass, `new A<>()`) is: (`new A<>() :A<`$\nu_1, \nu_3, \nu_4, \nu_6$`>`, $C_{\texttt{newA}}$)

with

$$C_{\texttt{newA}} = \{\ \nu_3 \lessdot \nu_5, \nu_4 \lessdot \nu_7, \nu_1 \lessdot \alpha_{\texttt{A}}^{\texttt{mt},1}\texttt{<}\nu_2, \nu_5\texttt{>}, \nu_2 \lessdot \gamma^{\texttt{m}}_{\alpha_{\texttt{A}}^{\texttt{mt},1}}\texttt{<}\nu_6, \nu_7\texttt{>}\ \}$$

The constraint set of the result of **TYPE** is given as

$$C_{\texttt{main}} = \{\ \nu_3 \lessdot \nu_5, \nu_4 \lessdot \nu_7, \nu_1 \lessdot \alpha_{\texttt{A}}^{\texttt{mt},1}\texttt{<}\nu_2, \nu_5\texttt{>}, \nu_2 \lessdot \gamma^{\texttt{sub}}_{\alpha_{\texttt{A}}^{\texttt{mt},1}}\texttt{<}\nu_6, \nu_7\texttt{>},$$
$$\texttt{myInteger} \lessdot \nu_1, \texttt{myInteger} \lessdot \nu_3, \texttt{myInteger} \lessdot \nu_4\ \}$$

The function **construct** adds no interfaces, as there is no call of abstract fields or methods. Therefore the result of **construct** is given as:

```
class Main<ν6> [C_Main] {
    ν6 main() {
        return new A<>()
               .mt(new myInteger(2), new myInteger(1), new myInteger(3));
    }
}
```

In **solve** $C_{\texttt{main}}$ is unified: The class declarations implies

$$\texttt{myInteger} \leq^* \alpha_{\texttt{A}}^{\texttt{mt},1}\texttt{<myInteger, myInteger>}$$

and

$$\texttt{myInteger} \leq^* \gamma^{\texttt{sub}}_{\alpha_{\texttt{A}}^{\texttt{mt},1}}\texttt{<myInteger, myInteger>}.$$

With the *adapt2*-rule follows from $\texttt{myInteger} \lessdot \nu_1, \nu_1 \lessdot \alpha_{\texttt{A}}^{\texttt{mt},1}\texttt{<}\nu_2, \nu_5\texttt{>}$:

$$\texttt{myInteger} \lessdot \nu_1, \nu_1 \lessdot \alpha_{\texttt{A}}^{\texttt{mt},1}\texttt{<myInteger, myInteger>},$$
$$\nu_2 \doteq \texttt{myInteger}, \nu_5 \doteq \texttt{myInteger}.$$

From this follows with the *subst*-rule

$$\texttt{myInteger} \lessdot \gamma^{\texttt{sub}}_{\alpha_{\texttt{A}}^{\texttt{mt},1}}\texttt{<}\nu_6, \nu_7\texttt{>}$$

and with the *adapt1*-rule:

$$\gamma^{\texttt{sub}}_{\alpha_{\texttt{A}}^{\texttt{mt},1}}\texttt{<myInteger, myInteger>} \doteq \gamma^{\texttt{sub}}_{\alpha_{\texttt{A}}^{\texttt{mt},1}}\texttt{<}\nu_6, \nu_7\texttt{>}.$$

With the *reduce1*- and the *swap*-rule we get:

$$\nu_6 \doteq \texttt{myInteger}, \nu_7 \doteq \texttt{myInteger}.$$

With the *subst*-rule follows from

$\texttt{myInteger} \lessdot \nu_3, \nu_3 \lessdot \texttt{myInteger}$ and $\texttt{myInteger} \lessdot \nu_4, \nu_4 \lessdot \texttt{myInteger}$

and from this with the *refl*-rule:

$$\nu_3 \doteq \texttt{myInteger}, \nu_4 \doteq \texttt{myInteger}.$$

The result of **solve** is given as:

$$\begin{aligned}\{\, &\texttt{myInteger} \lessdot \nu_1, \nu_1 \lessdot \alpha_{\texttt{A}}^{\texttt{mt},1}\texttt{<myInteger,myInteger>}\\ &\nu_2 \doteq \texttt{myInteger}, \nu_5 \doteq \texttt{myInteger}, \nu_6 \doteq \texttt{myInteger},\\ &\nu_7 \doteq \texttt{myInteger}, \nu_3 \doteq \texttt{myInteger}, \nu_4 \doteq \texttt{myInteger}\,\}\end{aligned}$$

The resulting Java class is given as:

```
class Main [ myInteger extends ν1,
             ν1 extends α_A^mt,1<myInteger, myInteger> ]
{
    myInteger main() {
        return new A<>().mt(new myInteger(2),
                            new myInteger(1),
                            new myInteger(3)); }
}
```

There is one remaining type variable ν_1, that is not used in a argument- or return-type of a method. Therefore ν_1 is no class-parameter of `Main`. The two remaining bounds of ν_1 are consistent. This means `main` is executable. The result of the execution is 4.

5 Summary

We have presented a type inference algorithm for a Java-like language. The algorithm allows to declare type-less Java classes independently from any environment. This allows separate compilation of Java classes without relying on type informations of other classes. The algorithm infers structural types, that are given as generated interfaces. The instances have to implement these interfaces.

References

[ADDZ05] Davide Ancona, Ferruccio Damiani, Sophia Drossopoulou, and Elena Zucca. Polymorphic bytecode: Compositional compilation for Java-like languages. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 26–37, New York, NY, USA, 2005. ACM.

[IPW01] Atsushi Igarashi, Benjamin C Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450, 2001.

[MM82] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4:258–282, 1982.

[Plü07] Martin Plümicke. Typeless Programming in Java 5.0 with Wildcards. In Vasco Amaral, Luís Veiga, Luís Marcelino, and H. Conrad Cunningham, editors, *5th International Conference on Principles and Practices of Programming in Java*, volume 272 of *ACM International Conference Proceeding Series*, pages 73–82, September 2007.

[Plü09] Martin Plümicke. Java type unification with wildcards. In Dietmar Seipel, Michael Hanus, and Armin Wolf, editors, *17th International Conference, INAP 2007, and 21st Workshop on Logic Programming, WLP 2007, Würzburg, Germany, October 4-6, 2007, Revised Selected Papers*, volume 5437 of *Lecture Notes in Artificial Intelligence*, pages 223–240. Springer-Verlag Heidelberg, 2009.

[Plü15] Martin Plümicke. More type inference in Java 8. In Andrei Voronkov and Irina Virbitskaite, editors, *Perspectives of System Informatics - 9th International Ershov Informatics Conference, PSI 2014, St. Petersburg, Russia, June 24-27, 2014. Revised Selected Papers*, volume 8974 of *Lecture Notes in Computer Science*, pages 248–256. Springer, 2015.

Statische Analysen von Online-Befragungen mit der Programmiersprache *liQuid*

Thomas M. Prinz, Linda Gräfe, Jan Plötner und Anja Vetterlein

Universitätsprojekt Lehrevaluation
Friedrich-Schiller-Universität Jena
07743 Jena, Deutschland
{Thomas.Prinz, Linda.Graefe, Jan.Ploetner, Anja.Vetterlein}@uni-jena.de

Zusammenfassung Befragungen in der empirischen Forschung und im Qualitätsmanagement werden zunehmend komplexer. Diese wachsende Komplexität entsteht vor allen Dingen durch die steigende Anzahl an Variablen und des sich erhöhenden Verzweigungsgrades in dem zur Befragung gehörenden Fragebogen. Das derzeit genutzte seitenbezogene, informelle Layout zur Beschreibung von Fragebögen wird dieser Komplexität nicht mehr gerecht.
Das Ziel unserer Forschungsarbeiten im Bereich der Psychoinformatik ist die Entwicklung einer domänenspezifischen Sprache zur Beschreibung umfangreicher Fragebögen. Für diese Sprache sollen Analysetechniken zur Verfügung gestellt werden, um die Stimmigkeit (im Sinne der Korrektheit) von Fragebögen zu überprüfen.
Wir schlagen in dieser Arbeit unsere Programmiersprache *liQuid* als domänenspezifische Sprache vor. Diese ermöglicht sowohl eine einfache, technische Beschreibung von Fragebögen als auch die Anwendung von Analysen. Als erstes Beispiel für solch eine Analyse zeigen wir, wie in *liQuid* Mehrfachabfragen ein und derselben Variablen durch statische Analysen erkannt werden können.

1 Einleitung

Befragungen mittels Fragebögen werden in der empirischen Forschung und im Qualitätsmanagement zunehmend komplexer. Die Komplexität vergrößert sich dabei vor allen Dingen in dem zur Befragung gehörenden Fragebogen hinsichtlich des Verzweigungsgrades (der Adaptivität) und der Anzahl an Variablen. Dies geht damit einher, dass immer mehr Befragungen computergestützt durchgeführt werden (bspw. die PISA-Studie[1]). Dadurch eröffnen sich neue Möglichkeiten der Implementierung der Fragebögen. Mit diesen neuen Möglichkeiten erhöht sich auch deren Komplexität. Dies wird auch im Universitätsprojekt Lehrevaluation (ULe) deutlich, in dem einzelne Lehrveranstaltungen sowie ganze Studiengänge an der Friedrich-Schiller-Universität Jena durch Fragebögen evaluiert werden.

Das derzeit in der Fragebogenkonstruktion genutzte, seitenbezogene und informelle Layout zur Beschreibung von Fragebögen wird dieser zunehmenden

[1] http://www.oecd.org/berlin/themen/pisa-studie/ (Juli 2017)

Komplexität kaum mehr gerecht, wie wir bspw. bei Befragungen von ULe feststellen mussten. Zeitgleich werden aber viele Fragebögen in diesem Format als Programme online zur Verfügung gestellt, um Befragungen schnell und einfach durchzuführen (bspw. SoSci Survey[2] und Survey Monkey[3]). Für solche elektronischen, mitunter umfangreichen Fragebögen gibt es derzeit jedoch keine Überprüfung auf deren technische Stimmigkeit, da sie generell eher informeller Natur sind.

Aus diesem Grund forschen wir an Modellen zur Formalisierung von Fragebögen und Methoden um diese Modelle statisch und dynamisch auf ihre technische Stimmigkeit zu überprüfen. Ein vielversprechender Ansatz zum Erreichen dieses Ziels ist die Entwicklung einer domänenspezifischen Programmiersprache zur Beschreibung komplexer Fragebögen. Für diese Sprache möchten wir Analysetechniken zur Verfügung stellen, um die Fragebögen auf Stimmigkeit zu testen. Der Begriff der *Stimmigkeit* ist dabei als Sammelbegriff für verschiedenste Probleme bei der Fragebogenkonstruktion zu verstehen.

In dieser Arbeit führen wir in die für die Fragebogenkonstruktion relevanten Begriffe ein und formalisieren sie, um sie in einem Modell eines Fragebogens vereinen zu können. Dabei verwenden wir nicht mehr ein seitenbezogenes Layout für Fragebögen, sondern greifen auf die Verwendung von Graphen zurück. Dies ermöglicht zum einen die Anwendung etablierter Analysemethoden der Graphentheorie. Zum anderen lassen sich im Gegensatz zum seitenbezogenen Layout durch Graphen die Verzweigungen in den Fragebögen intuitiver modellieren und strukturierter darstellen. Somit wird auch die Struktur eines Fragebogens erstmalig adäquat formalisiert.

Auf Basis dieses Modells führen wir die (konzeptionelle) domänenspezifische Programmiersprache *liQuid* ein. In dieser Arbeit verwenden wir einen eingeschränkten Umfang von *liQuid* bestehend aus verschiedenen Typen (Skalen), Variablen und Items (Fragestellungen). Diese werden genutzt, um Fragebögen zu programmieren. Da ein entstandenes Programm einem Kontrollflussgraphen gleicht, können Analysen aus dem Übersetzerbau erstmals auch auf Fragebögen angewendet werden. Als erstes Beispiel einer solchen Analyse zeigen wir, wie in *liQuid* Mehrfachabfragen einer Variablen erkannt werden können. Dazu werden die aus dem Übersetzerbau bekannten Begriffe der lebendigen Variablen und Datenflussanalysen genutzt.

Dieser Beitrag ist, wie folgt, aufgebaut: In den Grundlagen (Kapitel 2) definieren wir die von uns verwendeten Notationen von Digraphen und Pfaden. Anschließend führen wir Schritt für Schritt Begriffe der Fragebogenkonstruktion ein und formalisieren sie (Kapitel 3). Aus diesen Begriffen wird unsere Definition eines Fragebogens abgeleitet. Das Fragebogenmodell fließt direkt in Kapitel 4 in die Programmiersprache *liQuid* ein. In Programmen dieser Sprache können Mehrfachabfragen von Variablen erkannt werden. Dazu erläutern wir in Kapitel 5 zunächst die Problematik bei solch einer Mehrfachabfragung und wie diese mit Hilfe von lebendigen Variablen und Datenflussanalysen gelöst werden kann. Zum

[2] `https://www.soscisurvey.de/` (Juli 2017)
[3] `https://www.surveymonkey.com/` (Juli 2017)

Schluss fassen wir in Kapitel 6 diesen Beitrag zusammen und geben einen kurzen Ausblick auf zukünftige Arbeiten.

2 Grundlagen

Im Kontext dieser Arbeit ist der Begriff des *gerichteten Graphen* wichtig, um später Strukturen in Form von Graphen zu beschreiben.

Definition 1 (Gerichteter Graph). *Ein* gerichteter Graph *bzw.* Digraph G *setzt sich zusammen aus einer Menge* $N = N(G)$ *an* Knoten *und einer Menge* $E = E(G)$ *an* Kanten *von* G*.* E *bildet dabei eine Menge von geordneten Paaren,* $E \subseteq N \times N$*. Wir schreiben auch* $G = (N, E)$ *[3, S. 432 ff.].*

Durch die Verbindung der Knoten durch Kanten entstehen *Pfade* von einem Knoten zu einem anderen:

Definition 2 (Pfad). *In einem Digraphen* $G = (N, E)$ *heißt eine Sequenz* $P = (n_0, \ldots, n_m)$*,* $m \geq 0$*, von Knoten aus* N*,* $n_0, \ldots, n_m \in N$*, ein* Pfad*, sobald jeder Knoten mit seinem darauffolgenden Knoten durch eine Kante verbunden ist [4, S. 1180]:*

$$\forall\, 0 \leq i < m \colon (n_i, n_{i+1}) \in E \tag{1}$$

Ein Pfad heißt *azyklisch*, wenn alle Knoten des Pfads paarweise verschieden sind. Dem folgend heißt ein Digraph *azyklisch*, wenn jeder Pfad im Graph azyklisch ist. Anderenfalls beinhaltet der Graph eine Schleife und ist zyklisch. Wir beschreiben die Menge aller Pfade von einem Knoten a zu einem Knoten b durch $\mathcal{P}_{a \to b}$. Ein gerichteter Graph heißt *zusammenhängend*, wenn in seinem ungerichteten Pendant[4] von jedem Knoten zu jedem anderen Knoten ein Pfad existiert [10, S. 547].

3 Modell eines Fragebogens

Nachdem die Grundlagen für diesen Beitrag eingeführt wurden, führt dieses Kapitel die wichtigen Begriffe aus dem Bereich der Fragebogenkonstruktion ein. Dabei werden diese, wenn nötig, formalisiert. Anschließend wird aus diesen Begriffen unser Modell eines Fragebogens aufgebaut.

3.1 Variablen und Erhebungen

In der Messtheorie in den Sozial- und Verhaltenswissenschaften geht es um die Untersuchung der Eigenschaften von Objekten, im Speziellen von Menschen [2, S. 8]. In der Realität sind (psychologische) Eigenschaften (z. B. Intelligenz und Motivation) jedoch viel zu komplex und vielgestaltig, sodass diese beim Messen vergröbert und im Kontext der elektronischen Erfassung auch diskretisiert werden.

[4] Im gerichteten Graphen wird für jede Kante (a, b) ebenfalls eine Kante (b, a) eingefügt.

Demzufolge gibt es eine Abbildung von dem tatsächlichen Wertebereich einer Eigenschaft auf eine Menge diskreter Werte. Diese Menge an diskreten Werten nennen wir *Typ*. Dabei sollte eine Übertragung des *empirischen* Relativs auf einen *numerischen* Relativs innerhalb des Typs erfolgen [2, S. 17 ff.].

Durch die vorgenommene Diskretisierung besitzt jede Messung einer Eigenschaft bereits eine Ungenauigkeit. Aber nur mit Hilfe dieser Diskretisierung ist eine Messung überhaupt erst möglich. Somit ist die entstandene Inexaktheit aus praktischer Sicht unvermeidbar. Bei der Auswahl eines geeigneten Typs sollte auf die Reliabilität [9, S. 11] der Messungen geachtet werden.

Messen wir eine Eigenschaft eines Objekts *obj* mit Hilfe der Werte eines Typs $\mathcal{T}$, dann erhalten wir ein Paar (obj, val) mit $val \in \mathcal{T}$. Dieses Paar sagt aus, dass für *obj* bezüglich des Typs $\mathcal{T}$ der Wert *val* gemessen wurde. Führen wir nun Messungen an allen Objekten $\mathcal{O}$ durch, erhalten wir somit eine Abbildung von diesen Objekten auf den Typ. Diese Abbildung nennen wir eine *Variable*.

Definition 3 (Variable). *Eine* Variable *var ist eine linkstotale Abbildung von der Menge aller Objekte $\mathcal{O}$ auf einen Typ $\mathcal{T}$, $var\colon \mathcal{O} \mapsto \mathcal{T}$. Dabei schreiben wir für den Wertebereich der Variablen $\mathcal{T}(var)$.*

Mit einer Variablen werden also die Ausprägungen *einer* Eigenschaft (mit Hilfe einer Diskretisierung) von verschiedenen Objekten beschrieben [11, S. 19]. Die Variablenausprägungen sind vor den Messungen unbekannt. Manchmal sind auch theoretische Annahmen über diese Ausprägungen zugrunde gelegt, sollen jedoch empirisch überprüft werden. In beiden Fällen können wir davon ausgehen, dass sie durch verschiedene Messungen ermittelt werden müssen. Dazu Bedarf es an geeigneten *Messverfahren*. Ein Messverfahren ermittelt für ein Objekt *obj* und einer Menge von Variablen für jede dieser Variablen *var* ein Element $(obj, val) \in var$, $value \in \mathcal{T}(var)$. Sind die Variablen im Messverfahren geordnet, so ist das Messverfahren eine Abbildung von der Menge aller Objekte auf das kartesische Produkt der Typen aller im Messverfahren erhobenen Variablen.

Führen wir ein solches Messverfahren auf einer Menge von Objekten durch, dann sprechen wir von einer *Erhebung*.

Definition 4 (Erhebung). *Eine* Erhebung $\mathcal{H}$ *umfasst ein* Messverfahren $\mathcal{M}$, *eine Menge an* Objekten $\mathcal{O}$ *und eine Menge an* Variablen $\mathcal{V}$, $\mathcal{H} = (\mathcal{M}, \mathcal{O}, \mathcal{V})$. *Das Messverfahren $\mathcal{M}$ ist dabei eine Abbildung von der Menge der Objekte $\mathcal{O}$ auf das kartesische Produkt aller Typen aller Variablen:*

$$\mathcal{M}\colon \mathcal{O} \mapsto \prod_{v \in \mathcal{V}} \mathcal{T}(v) \tag{2}$$

Die Ergebnisse $\mathcal{R}$ der Erhebung $\mathcal{H}$ bilden sich aus der Vereinigung aller Ergebnisse aus den Messungen aller Objekte:

$$\mathcal{R} = \bigcup_{o \in \mathcal{O}} \mathcal{M}(o) \tag{3}$$

3.2 Items und Fragebögen

Im letzten Abschnitt haben wir uns allgemein mit den Begriffen befasst, die wichtig sind, um eine Befragung und einen Fragebogen zu definieren. In Hinblick auf Definition 4 ist eine *Befragung* (mittels Fragebogen) also eine spezielle *Erhebung* und der während der Befragung verwendete *Fragebogen* ein Messverfahren [11, S. 18]. Da ein Messverfahren eine Abbildung darstellt, können wir uns einen Fragebogen also einfach als Funktion, Algorithmus oder Programm vorstellen. Technisch erwartet dieses Programm als Eingabe einen Probanden (das Objekt) und führt zu einer Ausgabe, dem Messergebnis.

Wenn ein Fragebogen ein Programm ist, so können wir es auch mit Hilfe einer Programmiersprache beschreiben. Um diese zu entwickeln und bestenfalls Analysen auf diesen Programmen durchführen zu können, müssen wir jedoch einen Fragebogen noch näher betrachten. Nach Rost [11, S. 18] besteht ein Fragebogen aus sogenannten *Items*. Ein Item ist dabei eine konkrete Fragestellung oder Aufforderung inklusive der Antwortmöglichkeiten, die durch eine oder auch mehrere Variablen gegeben sind.

Definition 5 (Item). *Ein* Item $item = (\mathcal{Q}uest, \mathcal{V}ars)$ *besteht aus einer* Fragestellung $\mathcal{Q}uest(item)$ *und einer Menge abgefragter Variablen* $\mathcal{V}ars(item)$.

Manche Items müssen nicht von jedem Probanden abgefragt werden. Dies ist beispielsweise notwendig, wenn die Antwort auf ein Item die Beantwortung anderer Items ausschließt. Demzufolge benötigen wir *Bedingungen*, um Items vor Probanden zu verbergen:

Definition 6 (Bedingung). *Eine* Bedingung *über die Variablen* $v_0, \ldots, v_n$, $n \geq 0$, *ist eine linkstotale Abbildung vom kartesischen Produkt der Wertebereiche der Variablen (der Typen) auf die Menge* $\{wahr, falsch\}$*:*

$$\mathcal{T}(v_0) \times \cdots \times \mathcal{T}(v_n) \mapsto \{wahr, falsch\} \tag{4}$$

Aus der Testtheorie ist bekannt, dass der Aufbau eines Fragebogens Auswirkungen auf die Messergebnisse hat [9, S. 68 ff.]. Aus diesem Grund können wir einen Fragebogen nicht nur mit Hilfe einer Menge von Items und Bedingungen definieren, da Mengen ungeordnet sind und somit der Aufbau des Fragebogens daraus nicht ersichtlich wird. Es hat sich als vielversprechend herausgestellt, den Aufbau eines Fragebogens als gerichteten, azyklischen Graphen zu beschreiben:

Definition 7 (Fragebogengraph). *Ein* Fragebogengraph Q *(kurz* F-Graph*) besteht aus einem azyklischen, zusammenhängenden Digraphen* (I, E) *mit einer Menge von Items* I *und einer Menge von Kanten* E *zwischen den Items. Die im F-Graphen abgefragten Variablen bezeichnen wir mit* $\mathcal{V}ars(Q)$ *und ergeben sich zu* $\bigcup_{i \in I} \mathcal{V}ars(i)$*. Zusätzlich besitzt ein F-Graph noch eine linkstotale Abbildung,* $Cond$*, welche jeder Kante* $e \in E$ *eine Bedingung über einer Teilmenge der im F-Graph enthaltenen Variablen* $\mathcal{V}ars(Q)$ *zuweist. Zusammengefasst entspricht ein F-Graph demnach dem Tripel* $(I, E, Cond)$.

Wir nennen einen F-Graphen Q wohlgeformt*, wenn es genau ein Item ohne eingehende und genau ein Item ohne ausgehende Kanten gibt. Dabei heißt das Item ohne eingehende Kanten das* Start- *und das Item ohne ausgehende Kanten das* Enditem*.*

$$Q \text{ ist wohlgeformt} \tag{5}$$

$$\Longleftrightarrow$$

$$|\{i \in I(Q) \colon \triangleright i = \emptyset\}| = 1 \wedge |\{i \in I(Q) \colon i \triangleleft = \emptyset\}| = 1 \tag{6}$$

Ohne Beschränkung der Allgemeinheit gehen wir in den nachfolgenden Kapiteln von der Wohlgeformtheit von F-Graphen aus. Im allgemeinen Fall kann durch Hinzufügen eines expliziten Start- und Enditems für jeden F-Graphen seine Wohlgeformtheit sichergestellt werden.

Definition 7 beschreibt den *Aufbau* eines Fragebogens: Den F-Graphen. Auf Basis dieses Aufbaus lässt sich eine Messverfahren definieren, dass die gewünschten Variablen misst. Die Konstruktion dieses Messverfahrens entspricht der Ausführung des durch den F-Graphen beschriebenen Programms. Diese Ausführungssemantik ist jedoch aus Platzgründen nicht Teil der vorliegenden Arbeit.

4 Die konzeptionelle Programmiersprache *liQuid*

Mit Hilfe der im letzten Kapitel eingeführten Definition eines F-Graphen ist es uns nun möglich, konzeptionell eine domänenspezifische Sprache für Fragebögen zu entwerfen: *liQuid*. Aufgrund der im nächsten Kapitel beschriebenen ersten statischen Analyse auf einem Programm dieser Sprache, konzentrieren wir uns an dieser Stelle auf die Konzepte *Type*, *Variable* und *Item*. In Abbildung 1 ist ein Auszug aus der Grammatik des bisherigen Entwurfs von *liQuid* in Form eines Syntaxdiagramms zu sehen.

Das Konzept *Type* dient der Beschreibung von Typen. Ein Typ besitzt einen Bezeichner sowie seine beinhalteten Werte. Dabei werden diskrete Werte als Eingabe erwartet. Diese können entweder in Textform oder als Zahl angegeben werden. Die Definition von Typen in *liQuid* ist über den Befehl *Type* möglich (vgl. Produktionsregel $\langle type \rangle$ aus Abbildung 1).

Eine *Variable* wird, wie in anderen Programmiersprachen üblich, als eine Art Speicherzelle für Werte definiert. Dabei kennzeichnet ein Bezeichner eine Variable. Zusätzlich wird auch noch der Wertebereich der Variablen angegeben, sprich, der Typ. In *liQuid* werden Variablen durch den Befehl *Variable* definiert (vgl. Produktionsregel $\langle var \rangle$ aus Abbildung 1).

Das letzte Konzept *Item* erlaubt die Definition der im F-Graphen zu findenden Fragestellungen und die dabei aufgenommenen Variablen. Weiterhin wird durch das Item auch die Struktur des F-Graphen beschrieben. Dies erfolgt durch die Angabe der unmittelbar folgenden Items.

Zunächst muss jedem Item ein eindeutiger Bezeichner vergeben werden. Auf diesen folgt in geschweiften Klammern die Fragestellung $\langle question \rangle$, eine Kommaseparierte Auflistung von Variablenbezeichnern und eine (möglicherweise leere)

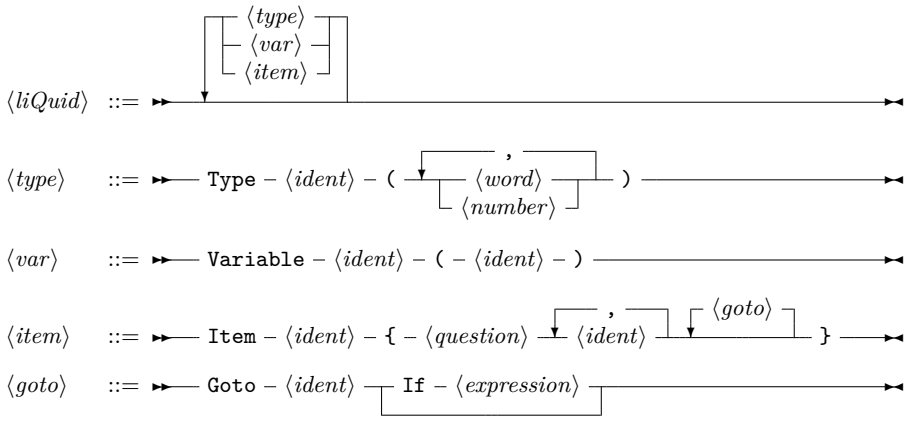


Abbildung 1. Auszug aus der Grammatik von *liQuid*

Auflistung von Sprungbefehlen, ⟨*goto*⟩. Ein Sprungbefehl besteht aus dem Befehl *Goto* und einer Sprungmarke, die einem Itembezeichner entsprechen muss. Optional kann einer Goto-Anweisung eine Bedingung nachgestellt werden. Die Definition eines Sprungbefehls beginnt mit der Angabe von *Goto* (vgl. Produktionsregel ⟨*goto*⟩ aus Abbildung 1). Die Definition eines Items startet mit dem Befehl *Item* (vgl. Regel ⟨*item*⟩ aus Abbildung 1).

Auf die Angabe der Produktionsregeln von ⟨*ident*⟩, ⟨*question*⟩, ⟨*number*⟩, ⟨*word*⟩ und ⟨*expression*⟩ wird aus Gründen der Übersichtlichkeit in Abbildung 1 verzichtet. Ein mittels der Sprache *liQuid* definiertes Beispielprogramm ist in Abbildung 2 zu sehen. Aus diesem Programm lässt sich der F-Graph aus Abbildung 3 gewinnen.

5 Auffinden von mehrfachen Abfragen der selben Variablen

Für die Einführung in die statische Analyse von Fragebögen auf deren (technische) Stimmigkeit untersuchen wir in dieser Arbeit das Auffinden von Mehrfachabfragen ein und derselben Variablen. Dafür müssen wir uns zunächst mit diesem Problem beschäftigen und herausfinden, warum es zu Schwierigkeiten während einer Messung kommen kann.

Nehmen wir dazu den F-Graphen eines *liQuid*-Programms aus Abbildung 3 an. Bei genauerer Betrachtung fällt auf, dass es in diesem Programm einige Variablen, wie die Variablen a und e, gibt, die an mehreren Stellen des F-Graphen abgefragt werden. Im Falle der Variablen e ist dies nicht problematisch, denn ein Proband kann e in jedweder Messung nur genau einmal ausfüllen. Hingegen kann die Variable a während einer Messung mehrmals ausgefüllt werden. Abbildung 4 illustriert solch einen Pfad im Fragebogen, auf dem a mehrmals abgefragt wird.

```
Type s1(1,2,3,4,5)
Type s2(5,4,3,2,1)

Variable a(s1)
Variable b(s2)
Variable c(s2)
Variable d(s1)
Variable e(s2)

Item i1 {"..."
  a
  Goto i2 If a == 4
  Goto i3
}
Item i2 {"..."
  e
  Goto i5 If e == 3
  Goto i6
}
Item i3 {"..."
  b,
  c,
  e
  Goto i4
}
Item i4 {"..."
  d
  Goto i7
}
Item i5 {"..."
  c
  Goto i9
}
Item i6 {"..."
  b
  Goto i7 If e == 2
  Goto i8
}
Item i7 {"..."
  b
  Goto i10
}
Item i8 {"..."
  a
  Goto i9
}
Item i9 {"..."
  d
  Goto i11
}
Item i10 {"..."
  b
  Goto i11
}
Item i11 {"..."
  c
}
```

Abbildung 2. Beispielfragebogen in *liQuid*

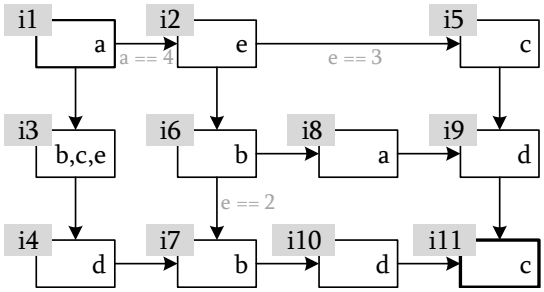


Abbildung 3. Beispiel eines F-Graphen

Solch eine mehrmalige Abfrage von Variablen ist aus den folgenden Gründen eine sowohl technische als auch inhaltliche Schwäche eines *liQuid*-Programms (F-Graphen) und sollte daher ausgeschlossen werden:

1. Bemerkt ein Proband das mehrfache Abfragen derselben Variablen, kann dies zu einer Minderung der Teilnahmemotivation und im schlimmsten Fall zum Abbruch der Befragung führen. Selbst wenn der Proband dies nicht bemerkt, wird der Fragebogen dennoch unnötig verlängert. Dies ist nicht im Sinne der Zumutbarkeit und Unverfälschbarkeit von Fragebögen [9, S. 22 f.].
2. Ein Proband könnte das zur zweiten (oder weiteren) Abfrage gehörende Item anders interpretieren und damit auch anders beantworten. Somit liegen sowohl aus inhaltlicher als auch technischer Sicht zwei unterschiedliche Werte einer Variablen vor. Sei nun beispielsweise angenommen, dass eine vorherige Sprunganweisung auf diese Variable bedingt. Dann kann es vorkommen, dass sich der Proband plötzlich in einem für ihn nicht vorgesehenen Pfad des F-Graphen befindet. Dies ist sowohl für die Interpretation der aufgenommenen Messwerte als auch für eine technische Realisierung ein undefinierter Zustand.

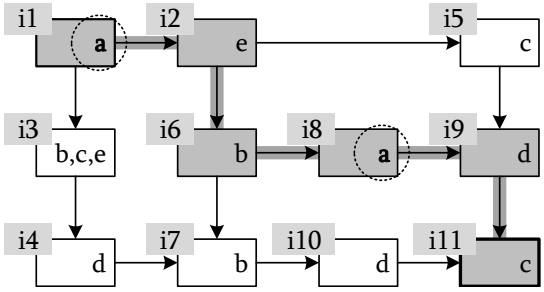


Abbildung 4. Ein Pfad im Fragebogen, in dem die Variable a mehrmals abgefragt wird

Um den zweiten Grund an einem Beispiel zu illustrieren, betrachten wir den F-Graphen aus Abbildung 3. Geht in diesem ein Proband den Pfad $(i1, i2, i6, i8)$ bis zum Item $i8$, dann wissen wir, dass der Proband der Variablen a den Wert 4 zugeordnet haben muss (aufgrund der Bedingung $a == 4$). Wenn er aber nun im Item $i8$ der Variablen a einen Wert *ungleich* 4 zuordnet, befindet er sich plötzlich an einer Position im F-Graphen, an der er nicht sein dürfte.

Aus den genannten Gründen ist die Mehrfachabfrage von Variablen während einer Durchführung einer Messung eines Probanden *nicht* erlaubt. Werden, wie bei Prä-Post-Messungen [6, S. 56 ff.], Variablen von Probanden zu unterschiedlichen Zeitpunkten abgefragt, so sollte dies entweder mittels zweier unterschiedlicher Messungen oder durch die Einführung zweier Variablen für die Prä- und Post-Messung modelliert werden.

Unter diesen Bedingungen ist eine mehrfache Abfrage einer Variablen formal definiert als das Auftreten einer Variablen in mindestens zwei Items, die auf dem selben Pfad von einem Start- zu einem Endknoten liegen. Diese Definition betrachtet eine Mehrfachabfrage aus statischer Sicht und schließt somit den dynamischen Ausschluss von Pfaden mittels nicht-erfüllter Bedingungen aus.

Definition 8 (Mehrfachabfrage von Variablen). *Sei $Q = (I, E, Cond)$ ein wohlgeformter Fragebogen mit dem Startitem s und dem Enditem e, sowie v, $v \in \mathcal{V}ars(Q)$, eine Variable des Fragebogens.*

Q beinhaltet eine Mehrfachabfrage *von v, wenn es einen Pfad vom Start- zum Enditem gibt, auf dem es mindestens zwei verschiedene Items gibt, die v beinhalten.*

$$\exists\, P \in \mathcal{P}_{s \to e}\ \exists\, i_1, i_2 \in P, i_1 \neq i_2 \colon v \in \mathcal{V}ars(i_1) \cap \mathcal{V}ars(i_2) \tag{7}$$

Um nun die Mehrfachabfragen in einem Programm zu erkennen, müssen wir laut Definition 8 alle Pfade vom Start- zum Enditem eines Fragebogens untersuchen. Dies können jedoch im schlechtesten Fall exponentiell viele sein, so dass sich eine Untersuchung auf Basis aller Pfade nicht lohnt und ggf. praktisch nicht möglich ist.

In unseren Untersuchungen haben wir jedoch einen Zusammenhang zwischen der Lebendigkeit von Variablen [8] in normalen Kontrollflussgraphen und dem

Mehrfachabfragen von Variablen in Fragebögen festgestellt. In einem gewöhnlichen Kontrollflussgraphen heißt eine Variable v für einen Knoten n *lebendig*, wenn der Wert von v in einem über n erreichbaren Knoten verwendet wird [8]. Dies ist eine klassische Definition aus der Optimierung zur Eliminierung nicht benötigten Programmcodes [1]. Beispielsweise ist die Variable a des Items $i1$ aus Abbildung 3 im Item $i2$ lebendig, da sie nochmals in Item $i8$ verwendet/definiert wird. Eine Variable v ist also *nur* nach i lebendig, wenn v *nach* i nochmals abgefragt wird. Das heißt, kann festgestellt werden, dass eine Variable v in einem direkt *nach* i folgenden Item *succ* lebendig ist, dann wird v offensichtlich mehrmals abgefragt – es liegt eine Mehrfachabfrage der Variablen v vor.

Um also die Mehrfachabfragen einer Variablen v in einem Fragebogen zu identifizieren, müssen wir lediglich die lebendigen Variablen für jedes Item bestimmen. Dies ist beispielsweise einfach durch eine Datenflussanalyse [5] mit Hilfe der Datenflussgleichung

$$LIVE(i) = DEF(i) \cup \bigcup_{succ \in \{s:\ (i,s) \in E\}} LIVE(succ) \tag{8}$$

möglich. Dabei ist $LIVE(x)$ die Menge der lebendigen Variablen des Items x und $DEF(x)$ die Menge der von x abgefragten Variablen. Die Lösung dieser Datenflussgleichung kann in jedem einschlägigen Buch über Übersetzerbau nachgelesen werden (bspw. [1]). Außerdem gehört sie zu den Datenflussproblemen, die sich in linearer Laufzeit für azyklische Graphen lösen lassen [7].

Nehmen wir an, die lebendigen Variablen wurden für jedes Item bestimmt. Anschließend muss für jedes Item i überprüft werden, ob eine von i abgefragte Variable in einem von i's direkten Nachfolgeitems lebendig ist. Ist dies der Fall, so gibt es auf diesem Pfad zum Enditem noch ein weiteres Item, dass diese Variable abfragt. Dann liegt ein Fehler vor. Anderenfalls ist für diese Variable sicher, dass sie zumindest ab diesem Item nicht noch einmal abgefragt wird.

Algorithmus 1 fasst den Ablauf zur Auffindung von Mehrfachabfragen von Variablen zusammen. In Abbildung 5 finden wir unseren Beispielfragebogen, für den die Informationen der lebendigen Variablen abgeleitet wurden. Zur Erinnerung, die lebendigen Variablen werden im inversen Graphen ausgehend vom Enditem berechnet, das heißt, in dem Graphen, in dem jede Kante umgedreht wurde. Wie in der Abbildung zu sehen ist, werden die in ein Item eingehenden Informationen der lebendigen Variablen für das Item vereinigt und durch die vom Item selbst abgefragten Variablen ergänzt (vgl. Gleichung 8).

Wie das Beispiel weiter zeigt, erkennt unser Algorithmus zuverlässig für das Item $i1$, dass die Variable a auf mindestens einem Pfad zum Enditem nochmals abgefragt wird. Dies wird anhand der von Item $i2$ eingehenden lebendigen Variablen $\{a, b, c, d, e\}$ und dem daraus resultierenden, nochmaligen Abfragen von a im Anschluss von $i1$ erkannt. Das gleiche gilt für die Variable b im Item $i6$, denn über den unteren Pfad wird die Variable b bereits abgefragt. Für die Variable e wird kein Fehler festgestellt, da es keinen Pfad vom Item $i1$ zum Item $i11$ gibt, auf dem e mehrfach abgefragt wird. Dies erkennt der Algorithmus korrekt.

Algorithm 1 Bestimmung von Mehrfachabfragen aller Variablen

Input: Fragebogen $Q = (I, E, Cond)$.
Output: Alle Mehrfachabfragen als Menge $Multiple \subseteq I \times \mathcal{V}ars(Q)$.

```
/** Berechne Datenflusssystem für lebendige Variablen.
    LIVE(i) ⊆ Vars(Q) sei die Menge der lebendigen Variablen für das Item i. **/
liveVariablesAnalysis(I, E)
/** Bestimme Mehrfachabfragen der Variablen für alle Items. **/
for all i ∈ I do
    for all succ ∈ {s: (i, s) ∈ I} do
        /** Bestimme Variablen, die im aktuellen Item i abgefragt werden und
            im direkten Nachfolger succ lebendig sind. **/
        alreadyDefined ← LIVE(succ) ∩ Vars(i)
        for all var ∈ alreadyDefined do
            /** Diese sind fehlerhaft. **/
            Multiple ← Multiple ∪ {(i, var)}
```

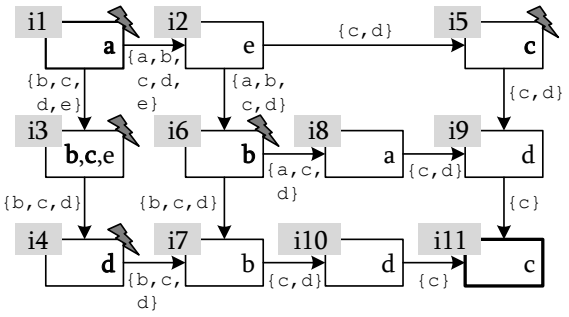


Abbildung 5. Statische Analyse des Beispielfragebogens

Das asymptotische Laufzeitverhalten des Algorithmus wird durch zwei Schritte bestimmt: (1) Die Bestimmung der lebendigen Variablen und (2) die Bestimmung der mehrfach definierten Variablen für jedes Item. Wir bereits erwähnt, zählt die Bestimmung der lebendigen Variablen mittels Datenflussanalyse (1) zu einem der einfachen Datenflussprobleme und ist in azyklischen Graphen in linearer Laufzeit $O(E)$ in Abhängigkeit der Kantenmenge möglich [7]. Die Bestimmung der mehrfach definierten Variablen (2) wird durch 3 Schleifen bestimmt: Die äußere, die mittlere und die innere For-All-Schleife. Die äußere und mittlere For-All-Schleife iterieren zusammen genau einmal über alle Kanten des Fragebogens. Demzufolge entspricht deren gemeinsames asymptotisches Laufzeitenverhalten $O(E)$. Die innere For-All-Schleife hingegen kann jedes Mal im Worst-Case über jede Variable des Fragebogens Q iterieren, $O(\mathcal{V}ars(Q))$. Es ergibt sich demnach eine Laufzeit von $O(E + \mathcal{V}ars(Q) * E)$ und demzufolge $O(\mathcal{V}ars(Q) * E)$.

6 Zusammenfassung

In diesem Beitrag führten wir zunächst wichtige Begriffe aus der Fragebogenkonstruktion für diese Arbeit ein. Daraus entwickelten wir ein Modell eines Fragebogengraphen, das einem azyklischen Digraphen entspricht und aus Items, Kanten und Kantenbedingungen besteht. Jedes Item umfasst eine Menge von Variablen, die durch das Item abgefragt werden.

Auf Basis dieses Fragebogenmodells entwickelten wir eine konzeptionelle Programmiersprache *liQuid*. Mit dieser können Fragebögen definiert, analysiert und ausgeführt werden. Als Beispiel einer Analyse wurde das Problem der Mehrfachabfrage von Variablen eingeführt und gezeigt, wie dieses mit Hilfe einer statischen Analyse mittels lebendiger Variablen gelöst werden kann.

In zukünftigen Arbeiten wird im Universitätsprojekt Lehrevaluation die Sprache *liQuid* um weitere Sprachkonstrukte, wie Repräsentationen und Regeln, ergänzt. Zudem soll die Semantik der Fragebogengraphen formalisiert sowie andere (technische) Probleme bei der Konstruktion von Fragebögen, wie die Nutzung von Variablen in Bedingungen vor deren Abfragung, analysiert und gelöst werden.

Literatur

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compiler: Prinzipien, Techniken und Werkzeuge. 2., aktualisierte Auflage, Pearson, München, Deutschland (2008)
2. Bühner, M., Ziegler, M.: Statistik für Psychologen und Sozialwissenschaftler. 1. Auflage, Pearson, München, Deutschland (2009)
3. Chartrand, G., Zhang, P.: Discrete Mathematics. 1 edn., Waveland Press, Inc., Long Grove, Illinois, USA (2011)
4. Cormen, T.H., Leiserson, C.E., Rivest, R., Stein, C.: Algorithmen - Eine Einführung. 4., durchgesehene und korrigierte Auflage, Oldenbourg, Munich, Germany (2013)
5. Hecht, M.S.: Flow Analysis of Computer Programs. Programming Languages Series, Vol. 5, 1 edn., Original von University of California, Elsevier North Holland, New York (1977)
6. Holling, H., Schmitz, B. (eds.): Handbuch Statistik, Methoden und Evaluation. Handbuch der Psychologie, Band 13. 1. Auflage, Hogrefe, Göttingen, Germany (2009)
7. Kam, J.B., Ullman, J.D.: Global Data Flow Analysis and Iterative Algorithms. Journal of the ACM 23(1), pp. 158–171 (Jan 1976)
8. Kennedy, K.: A Global Flow Analysis Algorithm. International Journal of Computer Mathematics 3(1-4), pp. 5–15 (1972)
9. Moosbrugger, H., Kelava, A. (eds.): Testtheorie und Fragebogenkonstruktion. 2., aktualisierte und überarbeitete Auflage, Springer, Berlin, Germany (2011)
10. Pahl, P.J., Damrath, R.: Mathematical Foundations of Computational Engineering: A Handbook. 1. Auflage, Springer, Berlin, Germany (2001)
11. Rost, J.: Lehrbuch Testtheorie – Testkonstruktion. Zweite, vollständig überarbeitete und erweiterte Auflage, Huber, Bern, Schweiz (2004)

Muli: Constraint-Programmierung in Java auf symbolischer JVM

Jan C. Dageförde and Herbert Kuchen

Westfälische Wilhelms-Universität Münster, D-48149 Münster
{dagefoerde,kuchen}@uni-muenster.de

Abstract. In der Praxis hat sich objektorientierte Programmierung mit Sprachen wie Java an vielen Stellen durchgesetzt, u. a. durch Features wie Kapselung von Struktur und Verhalten sowie Vererbung. Java ist allerdings nur bedingt geeignet zur Lösung von Suchproblemen, wie man sie beispielsweise bei der Personaleinsatzplanung oder der Produktionsplanung vorfindet.
Muli Lang (Münster Logic-Imperative Language) ergänzt die Programmiersprache Java um Möglichkeiten zur vereinfachten Lösung von Suchproblemen. Durch minimale sprachliche Ergänzungen führt Muli Lang logische Variablen und eingekapselte Suche ein.
Das zugehörige Laufzeitsystem *Muli Env* (Münster Logic-Imperative Environment) basiert auf einer symbolischen Java Virtual Machine (SJVM), welche Muli- und Java-Programme sowohl symbolisch als auch regulär zur Ausführung bringt. Das Laufzeitsystem wird außerdem um Komponenten wie Choice Points und Trail erweitert, welche von abstrakten Maschinen für logische Programmiersprachen, wie der Warren Abstract Machine (WAM), bekannt sind. Dadurch wird das Laufzeitsystem im Rahmen der eingekapselten Suche Backtracking-fähig, sodass auch nichtdeterministische Muli-Programme ausgeführt werden können.
Die enge Integration von Constraint-Programmierung und objektorientierter Programmierung macht Muli insbesondere für diejenigen Suchprobleme interessant, die aus Java-Anwendungen heraus berechnet werden und im Zeitverlauf inkrementell um Constraints ergänzt werden.

Keywords: Java Virtual Machine, Warren Abstract Machine, free variables, constraint solving, programming language integration

Zur Berechnung der softwaretechnischen Komplexität von einfachen objektorientierten Programmen

Marc Roßner und Michael Fothe

Friedrich-Schiller-Universität Jena, Fakultät für Mathematik und Informatik
Ernst-Abbe-Platz 2, 07743 Jena
{marc.rossner,michael.fothe}@uni-jena.de
https://www.fmi.uni-jena.de/

Abstract. Das Komplexitätsmaß nach Peter Rechenberg lässt sich auf einfache objektorientierte Programme übertragen. Solche Programme sind Gegenstand des Informatik-Unterrichts an Schulen. Die Softwaremetrik stellt eine Grundlage dafür dar, unterschiedliche Programme zu einer Aufgabe zu vergleichen.

Keywords: Softwaremetrik, Objektorientierung, Abituraufgaben

1 Einleitung

Im Thüringer Lehrplan für das Fach Informatik am Gymnasium ist für den Bereich der Selbst- und Sozialkompetenz unter anderem formuliert: "Der Schüler kann [...] Modelle und deren Implementierung beurteilen." [6, S. 20]. Für das Beurteilen zweier Implementierungen ein und desselben Modells wird eine Vergleichsgrundlage benötigt. In diesem Aufsatz wird dafür die Software-Metrik von Peter Rechenberg vorgeschlagen, und zwar in einer modifizierten Form, die sich auf einfache objektorientierte Programme bezieht. Rechenberg führte seine Metrik bereits 1986 mit dem Blick auf die Programmiersprachen Pascal (imperativ; strukturiertes Programmieren) und Modula-2 (modulares Programmieren) ein [5]. Die Gesamtkomplexität CC eines Programms ergibt sich demnach als Summe der Anweisungskomplexität SC, der Ausdruckskomplexität EC und der Datenkomplexität DC:

$$CC = SC + EC + DC \tag{1}$$

(siehe auch http://www.fmi.uni-jena.de/fmimedia/rossner_rechenberg.pdf). Es ist eine Forderung des Thüringer Lehrplans, "einfache Sachverhalte objektorientiert modellieren und implementieren" [6, S. 35] zu können. Sachverhalte, die Gegenstand der Abiturprüfung im Fach Informatik sind, dürften in diesem Sinne häufig als einfach anzusehen sein, sodass es lohnenswert erscheint, das modifizierte Verfahren auf solche Abituraufgaben anzuwenden. Exemplarisch wurden zwei Aufgaben aus Thüringen (Leistungsfach Informatik) auf verschiedenen Wegen in der Programmiersprache C++ gelöst, die Komplexität des

Quellcodes berechnet und die unterschiedlichen Implementierungen sowie deren Bewertungen verglichen. Ursprünglich waren die Abituraufgaben mit Turbo Pascal, Oberon-2 oder Java zu lösen.

Die Bewertungen der Programme können auch Ausgangspunkt für Diskussionen zu den Anforderungen in Informatikprüfungen sein.

2 Übertragen der Software-Metrik von Rechenberg auf einfache objektorientierte Programme und C++

In Konkretisierung von Festlegungen der EPA Informatik lassen sich die folgenden Kompetenzen angeben [3, S. 74]:

Die Schülerinnen und Schüler

a) erläutern Grundkonzepte der objektorientierten Modellierung (Objekt, Klasse, Vererbung, Polymorphie, Datenkapselung, Wiederverwendbarkeit),
b) modellieren Probleme, dokumentieren die Modelle und stellen die Modelle mit grafischen Mitteln dar,
c) analysieren, modifizieren und überprüfen eigene oder gegebene Modellierungen und
d) implementieren objektorientierte Modelle.

Unter einfachen objektorientierten Programmen sollen in diesem Aufsatz solche Programme verstanden werden, die sich ausschlielich auf die Grundkonzepte Objekt und Klasse beziehen. Diese Konzepte bieten die Möglichkeit des strukturierten Aufbaus von Softwareprojekten. Man kann Teillösungen in Klassen auslagern und diese in einem Hauptprogramm zur Gesamtlösung vereinen. Bei diesem Vorgehen kann die Implementierung der Methoden als quasi-prozedural angesehen werden; die Methoden können in der Software-Metrik wie gewöhnliche Prozeduren behandelt werden.

Für das Berechnen der Anweisungs- und der Ausdruckskomplexität können die Regelungen aus [5] unmittelbar übernommen werden. Das Berechnen der Datenkomplexität kann bei den Methoden so erfolgen, wie es bei Rechenberg für die modulare Programmierung beschrieben ist. Aus Sicht der Modul-Implementierung könnte die Schnittstelle als "äußerste Schicht" aufgefasst werden und alle Namen würden die Bewertung 3 erhalten. Andererseits muss man sich zur Implementierung des Moduls die Bedeutung der Namen ohnehin vergegenwärtigen, sodass man sie innerhalb des Moduls auch generell als lokal betrachten könnte. Es wird für C++ festgelegt:

Namen in Modulen werden nach der letzteren Variante mit 1 und erst bei deren Verwendung im Hauptprogramm mit 3 bewertet. Sollten Module ineinander geschachtelt sein, greift die Bewertung mit der Blocktiefendifferenz i.

Für Attribute soll gelten, dass den gekapselten Zustandsdaten, auf die alle oder fast alle Methoden der Klasse zugreifen, die Bewertung 3 zugewiesen wird unabhängig davon, ob innerhalb der Implementierung der Klasse oder im Hauptprogramm.

Alle Bezeichner des Namensraumes `std` sollen als Standardnamen gelten und somit keiner Bewertung unterliegen (z. B. `cout`, `cin`, Standarddatentypen).

Für die Verwendung von `cout` für die Kommandozeilenausgabe wird in Anlehnung an `printf(...)` festgelegt: Der gesamte als Ausgabe erzeugte Text wird als genau ein übergebener Parameter behandelt, unabhängig davon, aus wie vielen Teilen sich dieser zusammensetzt. Für die Bewertung gilt dann:

$$SC/EC(cout << '' \, Ergebnis :'' << ergebnis << endl;) = 1 + 1 \tag{2}$$

Die `return`-Anweisung besitzt in C++ einen Parameter. Daher wird festgelegt:

$$SC(return\, ergebnis;) = 1 + 1 \tag{3}$$

Der Operator der Ungleichheit `!=` besteht typografisch aus zwei Zeichen; es wird festgelegt:

$$EC(if(eingabe! = gesucht)) = 2 \tag{4}$$

3 Analysieren der Aufgabe "Plumpsack"

Das Spiel "Plumpsack" [2] ist mithilfe einer Ringliste zu realisieren. Die Ringliste wird durch ein Array implementiert. Es wurde eine prozedurale und eine objektorientierte Musterlösung erarbeitet. Die Tabellen 1 und 2 beinhalten für beide Lösungsvarianten die Bewertung des Teilprogramms zur Initialisierung einer Spielrunde.

Code	SC	EC	DC
`void spiel::init()`	nicht bewertet		
`{`			
`1 for (int i=0; i<anz_spieler; i++)`	3	2	4
`  {`			
`2  spielfeld[i]=i+1;`	1,5	4,5	4
`  }`			
`3 cout<<"Startaufstellung:"<<endl;`	2	2	0
`4 print();`	1	1	1
`}`			
Summe	7,5	9,5	9
CC		26	

Table 1. Bewertung der Methode zum Initialisieren des Spielfelds im objektorientierten Programm.

Code	SC	EC	DC
`void init(int spielfeld[], int anz_spieler)`	nicht bewertet		
`{`			
`1 for (int i=0; i<anz_spieler; i++)`	3	2	3
`{`			
`2 spielfeld[i]=i+1;`	1,5	4,5	3
`}`			
`3 cout<<"Startaufstellung:"<<endl;`	2	2	0
`4 print(spielfeld, anz_spieler);`	3	3	5
`}`			
Summe	9,5	11,5	11
CC		32	

Table 2. Bewertung der Prozedur zum Initialisieren des Spielfelds im prozeduralen Programm.

Beide Teilprogramme unterscheiden sich in der Zeile 4 beim Aufruf der print-Anweisung. Im prozeduralen Programm werden die Datenstrukturen von außen als Parameter übergeben. Im objektorientierten Programm wird direkt auf die in der Klasse gekapselten Datenstrukturen zugegriffen. Dem Bewertungs-Vorteil der fehlenden Parameter bei der Anweisungs- und Ausdruckskomplexität steht in der Objektorientierung die höhere Datenkomplexität gegenüber. Die Köpfe von Unterprogramm und Methoden sind unterschiedlich, werden jedoch nicht bewertet.
Die Erzeugung von Zufallszahlen wurde auf zwei Wegen realisiert; die Tabellen 3 und 4 geben Musterlösungen einschließlich Bewertung an. Beide Varianten unterscheiden sich in der Gesamtkomplexität nur wenig. Auch an diesem Beispiel zeigt sich somit, dass verschiedene Arten der Realisierung zu vergleichbaren Bewertungen führen können.

Code	SC	EC	DC
`bool gefangen()`	nicht bewertet		
`{`			
`time_t zeit;`	nicht bewertet		
`1 time(&zeit);`	2	2	4
`2 srand((unsigned int)zeit);`	2	2	4
`3 if (rand()%2==0)`	4	5	3
`4 return false;`	3	0	0
`5 else`			
`6 return true;`	3	0	0
`}`			
Summe	14	9	11
CC		34	

Table 3. Bewertung der ersten Variante zum Erzeugen einer Zufallszahl.

Code	SC	EC	DC
`bool gefangen()`	nicht bewertet		
`{`			
`  1 srand(static_cast<int>(time(NULL)));`	4	5	9
`  2 if (rand()%2==0)`	4	5	3
`  3  return false;`	3	0	0
`  4 else`			
`  5  return true;`	3	0	0
`}`			
Summe	14	10	12
CC		36	

Table 4. Bewertung der zweiten Variante zum Erzeugen einer Zufallszahl.

In der Tabelle 5 sind die Bewertungen der vollständigen prozeduralen und objektorientierten Programme zum Spiel "Plumpsack" angegeben.

	prozedural	objektorientiert
Anweisungskomplexität	121,75	112,75
Ausdruckskomplexität	118	110
Datenkomplexität	122	136
Gesamtkomplexität	361,75	358,75

Table 5. Bewertung von Programmen zur Aufgabe "Plumpsack".

Die Übertragung der prozeduralen in die objektorientierte Lösung erfolgte lediglich durch einige redaktionelle Umgruppierungen im Quelltext und dessen Anpassung an die Syntax der Objektorientierung. Beide Programme besitzen die gleiche Anzahl bewerteter Quellcodezeilen; nur sieben der bewerteten Codezeilen mussten geändert werden. Die Gesamtkomplexitäten der beiden Programme sind nahezu gleich; der Unterschied macht nur rund 1 % aus. Aus dieser Beobachtung lässt sich ableiten, dass man es in der Abiturprüfung dem Prüfungsteilnehmer überlassen kann, ob er eine Aufgabe prozedural oder objektorientiert löst (bei Beschränkung auf die Grundkonzepte Klasse und Objekt). Die Gesamtkomplexitäten der erarbeiteten Programme sind praktisch gleich.

4 Analysieren der Aufgabe "Liste"

Eine einfach verkettete Liste ist mit Hilfe eines Arrays zu realisieren. Je Listenelement ist ein Buchstabe als Inhalt und ein Integer-Wert als Verweis auf das nächste Element zu speichern (Genaueres siehe [1]). Die folgenden drei objektorientierten Varianten wurden implementiert und bewertet:

– **Q**uick **a**nd **D**irty: zwei eindimensionale Arrays des jeweiligen Datentyps

- **2D**-Array: ein zweidimensionales string-Array mit Typumwandlung (da negative Werte möglich sind)
- **S**low **a**nd **C**lean: ein eindimensionales struct-Array

Die Unterschiede in den drei Musterlösungen sollen am Beispiel der `search`-Methode deutlich gemacht werden, welche die Position des gesuchten Elements als Rückgabewert liefert bzw. -1, wenn das Element in der Liste nicht enthalten ist; siehe die Tabellen 6, 7 und 8.

Code	SC	EC	DC
`int liste::search(string element)`	nicht bewertet		
`{`			
`1 int finished=anfang;`	1	0	4
`2 while (finished!=1)-`	3	2	1
`{`			
`3 if (inhalt[finished]==element) return finished;`	9	3	7
`4 else finished=zeiger[finished];`	2,25	2	4
`}`			
`5 return -1;`	2	0	0
`}`			
Summe	17,25	7	16
CC	40,25		

Table 6. Lösungsvariante QaD.

Code	SC	EC	DC
`int liste::search(string element)`	nicht bewertet		
`{`			
`1 int finished=anfang;`	1	0	4
`2 while (finished!=1)-`	3	2	1
`{`			
`3 if (zweidfeld[finished][0]==element) return finished;`	9	5	7
`4 else finished=inInt(zweidfeld[finished][1]);`	9	6	5
`}`			
`5 return -1;`	2	0	0
`}`			
Summe	24	13	17
CC	54		

Table 7. Lösungsvariante 2D.

Unterschiedlich sind jeweils die Zeilen 3 und 4. Die SaC-Lösung hat gegenüber der QaD-Lösung wegen der Verwendung des Punktoperators eine höhere Ausdruckskomplexität. Die zusätzliche Kapselung bei QaD macht sich bei der Datenkomplexität bemerkbar. Bei der 2D-Realisierung erfolgt eine doppelte Indizierung,

Code	SC	EC	DC
`int liste::search(string element)`	nicht bewertet		
`{`			
`1 int finished=anfang;`	1	0	4
`2 while (finished!=1)-`	3	2	1
`{`			
`3 if (eindfeld[finished].value==element) return finished;`	9	4	10
`4 else finished=eindfeld[finished].next;`	2,25	3	7
`}`			
`5 return -1;`	2	0	0
`}`			
Summe	17,25	9	22
CC	48,25		

Table 8. Lösungsvariante SaC.

was eine Verdopplung der Bewertung zur Folge hat. Zusätzlich erfolgt ein Funktionsaufruf bzw. eine Schachtelung zur Datentypkonvertierung, woraus eine höhere Bewertung folgt. Wegen der höheren Anzahl an Fehlerquellen und der damit erhöhten Anforderung an die Konzentration beim Implementieren wird auch diese Unterscheidung als gerechtfertigt angesehen. Die Bewertungen der vollständigen Musterlösungen sind der Tabelle 9 zu entnehmen.

	QaD	2D	SaC
Anweisungskomplexität	339,5	377	338,5
Ausdruckskomplexität	200	258	221,5
Datenkomplexität	308	312	361
Gesamtkomplexität	847,5	947	921

Table 9. Bewertung von drei Musterlösungen zur Aufgabe "Liste".

Die drei Musterlösungen unterscheiden sich im Quellcode in nur 19 Zeilen. Die meisten Zeilen sind gleich, die drei Hauptprogramme sind sogar vollständig gleich. Im Hauptprogramm waren das Wort "Informatik" buchstabenweise in der Datenstruktur zu speichern und alle Funktionen daran zu testen. Der insert-Aufruf erfolgt also zehnmal hintereinander. Man könnte dies im Sinne von copy-and-paste abschwächend in der Komplexität berücksichtigen. Rechenberg trifft dazu keine Aussage und die Codezeilen sind deshalb in der Bewertung auch voll berücksichtigt. Im Endergebnis unterscheiden sich die günstigste und die ungünstigste Variante in der Gesamtkomplexität um 11,7 %. Wenn man es dem Prüfungsteilnehmer überlässt, mit welcher Datenstruktur er die Aufgabe löst, muss man damit rechnen, dass sich die Gesamtkomplexitäten des entwickelten Quellcodes in der Größenordnung 10% voneinander unterscheiden. Diese Größenordnung erscheint auch in einer Abiturprüfung vertretbar.

5 Ausblick

Mit dem vorliegenden Aufsatz wurde ein Verfahren vorgestellt, mit dem sich die softwaretechnische Komplexität von einfachen objektorientierten Programmen berechnen lässt. Die Verwendung von weiteren Grundkonzepten über Klasse und Objekt hinaus würde der Komplexität eines objektorientierten Programms gegenüber eines prozeduralen eine neue Qualität verleihen. Die Rechenbergschen Festlegungen sind in dem Fall nicht unmittelbar anwendbar. Dies könnte Gegenstand weiterer Untersuchungen sein.

Die für diesen Aufsatz erarbeiteten Musterlösungen und deren Komplexitäten liefern Hinweise darauf, dass es Prüfungsteilnehmern freigestellt werden kann, ob sie prozedural oder objektorientiert programmieren und welche Datenstruktur von ihnen eingesetzt wird (sofern die Datenstruktur als prinzipiell geeignet anzusehen ist). Die Komplexitäten der jeweiligen Programme unterscheiden sich nur in einer akzeptablen Größenordnung.

Nachdenklich stimmt der Unterschied in den Gesamtkomplexitäten von Programmen zu den beiden Abituraufgaben. Die Programme zum "Plumpsack" besitzen eine Gesamtkomplexität von ungefähr 360, die Programme zur Liste eine Gesamtkomplexität von rund 850-950, obwohl mit beiden Aufgaben die gleiche Anzahl von Bewertungseinheiten erreichbar war (jeweils 30 BE von 60 BE für die gesamte Abiturpüfung). Vor vorschnellen Urteilen ("Die eine Aufgabe war viel schwerer und verlangte auch viel mehr Bearbeitungsaufwand als die andere.") ist jedoch zu warnen, denn bei der Berechnung der Gesamtkomplexität wird nur das Endprodukt eingeschätzt. Schwierigkeiten und Aufwand in der Phase der Modellierung werden nicht bewertet. Selbstverständlich können die unterschiedlichen Komplexitäten Anlass zur Kommunikation zur Schwierigkeit und dem Programmieraufwand von Abituraufgaben sein.

References

1. Freistaat Thüringen: Abiturprüfung 2007 Leistungsfach Informatik (Haupttermin).
2. Freistaat Thüringen: Abiturprüfung 2008 Leistungsfach Informatik (Haupttermin).
3. Fothe, Michael: Kunterbunte Schulinformatik - Ideen für einen kompetenzorientierten Unterricht in den Sekundarstufen I und II. LOG IN Verlag, Berlin 2010.
4. Gloeckner, Christian: Entwurf und Implementierung elementarer Algorithmen im Informatik-Unterricht der Sekundarstufen I und II. Wissenschaftliche Hausarbeit zum ersten Staatsexamen an Gymnasien an der Friedrich-Schiller-Universität Jena (2016).
5. Rechenberg, Peter: Ein neues Maß für die softwaretechnische Komplexität von Programmen. In: Informatik Forschung und Entwicklung (1986) 1: 26-37.
6. Thüringer Ministerium für Bildung, Wissenschaft und Kultur: Lehrplan für den Erwerb der allgemeinen Hochschulreife Informatik (2012).

Flexibler Abbruch von Anweisungen in LISP FlexiPI

Christian Heinlein

Hochschule Aalen – Technik und Wirtschaft
christian.heinlein@hs-aalen.de

Erweiterte Zusammenfassung

Viele imperative Programmiersprachen bieten Anweisungen, mit denen umschließende Anweisungen vorzeitig beendet werden können, zum Beispiel (Syntax gemäß C, Java u. ä.):

- `continue` bricht den aktuellen Durchlauf der umschließenden Schleife ab und beginnt ggf. sofort mit dem nächsten Durchlauf (sofern die Schleifenbedingung noch erfüllt ist).
- `break` bricht die gesamte umschließende Schleife ab.
- `break` mit einer Marke (label) bricht in Java die umschließende Anweisung ab, die mit dieser Marke gekennzeichnet ist. (Dabei muss es sich nicht unbedingt um eine Schleife handeln. Obwohl dies stark an `goto` in C erinnert, sind damit nur Sprünge von innen nach außen und somit kein beliebiger „Spaghetti-Code“ möglich.)
- `return` bricht die umschließende Prozedur (Funktion, Methode, Konstruktor o. ä.) ab. (Darüber hinaus kann `return` einen Resultatwert zurückgeben; dieser Aspekt wird hier jedoch nicht betrachtet.)
- `throw` bricht eine unbestimmte Anzahl umschließender Anweisungen und ggf. Prozeduren ab, bis die von dieser Anweisung geworfene Ausnahme mittels `try-catch` wieder aufgefangen wird.

Damit ist `throw` offensichtlich das allgemeinste und „stärkste“ Konstrukt dieser Art, sodass sich die anderen prinzipiell mit dessen Hilfe simulieren lassen. Um beispielsweise eine Schleife vorzeitig beenden zu können, kann man diese in eine `try-catch`-Anweisung einbetten und dann den Abbruch mittels `throw` realisieren (Syntax gemäß C++):

```
// Leere Dummy-Klasse zur Verwendung als Ausnahme.
class Break {};

try {
    // Schleife.
    for (int i = 1; i <= 10; i++) {
        ......
```

```
            // Vorzeitiger Abbruch bei i == 5.
            if (i == 5) throw Break();
            ......
        }
    }
    catch (Break) {
        // Leerer catch-Block.
    }
```

Tatsächlich wird in der Sprachdefinition von Modula-3 die Semantik von `EXIT` und `RETURN` (wobei `EXIT` dem `break` in C-artigen Sprachen entspricht) auf die von `RAISE` (entspricht `throw`) zurückgeführt.

Obwohl derartige „konzeptuelle Sparsamkeit“ für die formale Beschreibung einer Programmiersprache angebracht sein mag, würde sie beim praktischen Einsatz auf Kosten des Benutzers gehen, d. h. dieser möchte neben dem allgemeinsten Konstrukt `throw` auch die spezielleren Anweisungen wie `break` und `return` nicht missen.

In einer syntaktisch erweiterbaren Programmiersprache wie MOSTflexiPL (vgl. http://flexipl.info) verhält sich dies jedoch anders: Hier stehen konzeptuelle Sparsamkeit und Benutzerfreundlichkeit nicht im Widerspruch zueinander, weil sich bequem verwendbare Spezialkonstrukte jederzeit leicht unter Verwendung eines allgemeineren Mechanismus definieren lassen. Tatsächlich ist konzeptuelle Sparsamkeit eines der Grundprinzipien dieser Sprache:

- Es gibt beispielsweise nur eine einzige vordefinierte Fallunterscheidung, mit deren Hilfe weitere Verzweigungsarten wie z. B. `switch-case` definiert werden können.
- Ebenso gibt es nur einen einzigen vordefinierten Schleifentyp, mit dessen Hilfe sich beliebige weitere – kopfgesteuert, fußgesteuert, Zählschleife, Mengenschleife etc. – leicht definieren lassen.
- Es gibt nur eine „Handvoll“ Grundkonstrukte für parallele Programmierung, mit deren Hilfe sich zahllose Spezialkonstrukte wie z. B. Monitore, Futures, Pipes, MVars etc. realisieren lassen.
- Usw.

Dementsprechend gibt es in MOSTflexiPL auch nur eine einzige allgemeine Sprung- bzw. Abbruchanweisung, die ähnlich mächtig ist wie `throw`, tatsächlich aber sogar noch weitere Anwendungsmöglichkeiten bietet:

- Wenn ein Zweig einer parallelen Ausführung `...||...` eine Anweisung abbricht, die die parallele Ausführung umschließt, muss dadurch zwangsläufig auch der andere Zweig abgebrochen werden.
  Anders als bei den bisher betrachteten Fällen, erfolgt der Abbruch dieses anderen Zweigs dann nicht von „innen“, sondern von „außen“.
- Die Marken, mit denen abbrechbare Anweisungen gekennzeichnet werden können, sind „Bürger erster Klasse“, die z. B. als Parameter an Hilfsfunktionen übergeben oder in Variablen gespeichert werden können.

- Damit muss sich eine Anweisung wie `break` zur vorzeitigen Beendigung einer Schleife nicht mehr zwingend im lexikalischen Gültigkeitsbereich der Schleife befinden, sondern kann – wie jeder andere Code, der innerhalb der Schleife ausgeführt werden soll – auch in eine Hilfsfunktion verlagert werden, was in gängigen Programmiersprachen nicht möglich ist. (Wenn man `break` dort allerdings mittels `throw` simuliert, ist es interessanterweise doch möglich!)
- Wenn die Marke einer Anweisung in einer geeigneten Variablen gespeichert wird, kann eine parallel ausgeführte Anweisung diese ebenfalls benutzen, um die markierte Anweisung von „außen“ abzubrechen.
- Außerdem kann es natürlich vorkommen, dass eine Anweisung abgebrochen werden soll, obwohl sie bereits beendet ist. In diesem Fall ist die Abbruchanweisung einfach wirkungslos.

Aufgrund der vielfältigen Verwendungsmöglichkeiten, insbesondere im Zusammenspiel mit parallelen Ausführungen, ist die implementierungstechnische Umsetzung der beschriebenen Abbruchanweisung relativ komplex und erfordert sorgfältige Synchronisation, um fehlerhafte Ausführungen (race conditions) einerseits und Verklemmungen andererseits zu vermeiden.

Um Abbrüche von außen überhaupt portabel (d. h. ohne direkte Verwendung spezieller Betriebssystemmechanismen) zu ermöglichen, wird kooperatives Multitasking verwendet: Jeder Thread des MOSTflexiPL-Laufzeitsystems überprüft regelmäßig einen ihm zugeordneten Abbruchindikator, der von außen, d. h. von einem anderen Thread, gesetzt werden kann. Wenn ein Thread gerade stillsteht, weil er z. B. auf eine Benutzereingabe oder die Freigabe eines Semaphors wartet, sind diese Blockierungszustände durch geeignete Vorkehrungen so gestaltet, dass sie ebenfalls jederzeit von außen unterbrochen werden können.

Auf diese Weise kann das gesamte MOSTflexiPL-Laufzeitsystem ausschließlich unter Verwendung von Mitteln der C++11-Standardbibliothek realisiert werden.

An Online Scripting Language for Teaching Combinatorial Scientific Computing

M. Ali Rostami and H. Martin Bücker

Friedrich Schiller University Jena, Germany,
a.rostami@uni-jena.de, martin.buecker@uni-jena.de

Combinatorial Scientific Computing (CSC) [8, 9] consists of developing combinatorial models for problems in scientific computing, designing algorithms for the solution of the resulting combinatorial subproblems, and engineering corresponding software. CSC plays a prominent role in well-known areas like sparse matrix computations [3], mesh generation [4], and parallel computing [5]. However, CSC is also ubiquitous in a less common field called automatic or algorithmic differentiation [6] where computations involving Jacobian or Hessian matrices are transformed into a rich set of graph problems. Teaching CSC is tricky since the student should not only understand the combinatorial problem and the corresponding scientific computing problem, but also the intimate connection between these two problem representations. EXPLAIN [2, 10, 7, 1, 11] is a collection of interactive software modules currently developed at Friedrich Schiller University Jena to help teaching CSC in the classroom. Each module shows, side by side, a matrix encoding a problem from scientific computing and its related graph problem. So, the student can observe and analyze modifications that result during the solution of a problem in both matrix and graph views simultaneously.

We propose a scripting language on top of the Javascript library D3 (Data-Driven Documents) for implementing a new EXPLAIN module. This scripting language encapsulates D3 commands such that the student can edit and color both graph and matrix without the need to think of the underlying system. At the same time, some basic mathematical functions are added for the sake of simplicity. Consider the so-called column compression module in EXPLAIN as an example. This module implements a particular CSC problem involving sparse Jacobian matrices and a corresponding graph coloring problem. More precisely, we focus on a single step of a typical graph coloring algorithm. The following script assigns a color to a vertex of a given graph that is different from the colors of its neighbors:

```
1 var ns = neighbors(current);
2 var col_ns = get_colors(ns);
3 var new_col = min(diff(colors,col_ns));
4 color_vertex(current,new_col);
5 color_column(current, new_col);
```

Here, the function `neighbors` returns a set of all neighbors of a given vertex. Similarly, the function `get_colors` returns the colors of a given set of vertices. The mathematical functions `min` and `diff` compute the minimum of a set and

the difference of two given sets, respectively. In this particular CSC problem, the vertex of the graph represents a column of a sparse Jacobian matrix; see [2] for more details. The last two lines of this script not only color the vertex of the graph, but also the column of the Jacobian that is represented by this vertex.

Furthermore, we implement an online editor for this scripting language which is available immediately adjacent to the graph and matrix views. Figure 1 shows these two views together with the editor for the column compression module on the right. The student can directly edit and run the code written in this editor. The aim of this feature is to help students to interactively develop their own new module without the need of extensive knowledge of D3. There is a set of global variables which the student can edit in the upper input box of the editor labeled "Globals." Some of these global variables store important default values like the graph type, the colors, and the starting matrix. The values of other global variables can be assigned in the same way. There is another input box labeled "Code" that defines a single step of an iterative algorithm. In this graph coloring example, the order of processing the vertices is important. This order can be selected from a default list or is specified by the variable `order`. An animation of this graph algorithm can be controlled by various buttons below the editor. This graph algorithm can also be carried out by clicking on the graph vertices.

The implementation first evaluates the wrapper functions for D3 and then passes the contents of this editor to the `eval` function of Javascript. The contents of the editor can be sent either line by line or as a whole function, depending on an option selectable by the student.

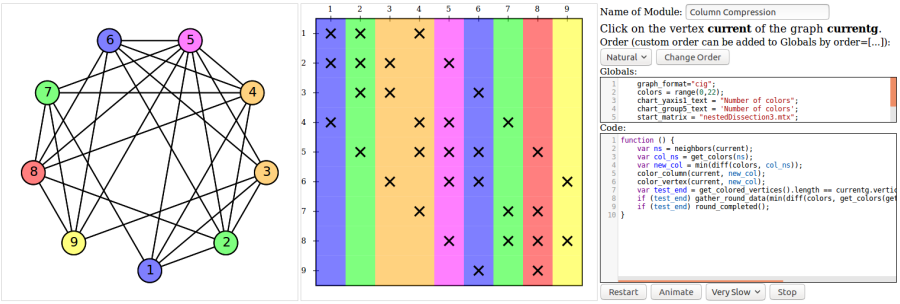


Fig. 1. The online editor of the corresponding module is visualized next to the graph and matrix. The code contained in the editor is written in a simple scripting language. The student first specifies the order of processing the vertices and this code is then executed using that order.

Keywords: combinatorial scientific computing, graph algorithms, education, scripting language

References

1. Bücker, H.M., Rostami, M.A.: Interactively exploring the connection between bidirectional compression and star bicoloring. In: Koziel, S., Leifsson, L., Lees, M., Krzhizhanovskaya, V.V., Dongarra, J., Sloot, P.M.A. (eds.) International Conference on Computational Science, ICCS 2015 — Computational Science at the Gates of Nature, Reykjavík, Iceland, June 1–3, 2015. Procedia Computer Science, vol. 51, pp. 1917–1926. Elsevier (2015)
2. Bücker, H.M., Rostami, M.A., Lülfesmann, M.: An interactive educational module illustrating sparse matrix compression via graph coloring. In: 2013 International Conference on Interactive Collaborative Learning (ICL), Proceedings of the 16th International Conference on Interactive Collaborative Learning, Kazan, Russia, September 25–27, 2013. pp. 330–335. IEEE, Piscataway, NJ (2013)
3. Duff, I., Erisman, A.M., Reid, J.: Direct Methods for Sparse Matrices. Numerical Mathematics and Scientific Computation Series, Oxford University Press (2017)
4. Edelsbrunner, H.: Geometry and Topology for Mesh Generation. Cambridge Monographs on Applied and Computational Mathematics, Cambridge University Press (2001)
5. Grama, A., Karypis, G., Kumar, V., Gupta, A.: Introduction to Parallel Computing. Pearson, 2nd edn. (2003)
6. Griewank, A., Walther, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. SIAM, Philadelphia, PA, 2nd edn. (2008)
7. H. M. Bücker, M. A. Rostami: Interactively exploring the connection between nested dissection orderings for parallel Cholesky factorization and vertex separators. In: IEEE 28th International Parallel and Distributed Processing Symposium, IPDPS 2014 Workshops, Phoenix, Arizona, USA, May 19–23, 2014. pp. 1122–1129. IEEE Computer Society, Los Alamitos, CA, USA (2014)
8. Hendrickson, B., Pothen, A.: Combinatorial Scientific Computing: The Enabling Power of Discrete Algorithms in Computational Science, pp. 260–280. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
9. Naumann, U., Schenk, O.: Combinatorial Scientific Computing. Chapman & Hall/CRC, 1st edn. (2012)
10. Rostami, M.A., Bücker, H.M.: Interactive educational modules illustrating sparse matrix computations and their corresponding graph problems. In: für Informatik, G. (ed.) Informatiktage 2014, Fachwissenschaftlicher Informatik-Kongress, 27. und 28. März 2014, Hasso Plattner Institut der Universität Potsdam, GI-Edition: Lecture Notes in Informatics (LNI) – Seminars, vol. S–13, pp. 253–256. Köllen Druck+Verlag GmbH, Bonn (2014)
11. Rostami, M.A., Bücker, H.M.: An educational module illustrating how sparse matrix-vector multiplication on parallel processors connects to graph partitioning. In: Hunold, S., Costan, A., Giménez, D., Iosup, A., Ricci, L., Gómez Requena, M.E., Scarano, V., Varbanescu, A.L., Scott, S.L., Lankes, S., Weidendorfer, J., Alexander, M. (eds.) Euro-Par 2015: Parallel Processing Workshops, Euro-Par 2015 International Workshops, Vienna, Austria, August 24–28, 2015, Revised Selected Papers. Lecture Notes in Computer Science, vol. 9523, pp. 135–146. Springer, Cham, Switzerland (2015)

Hypertree Decomposition for Constraint Programming in Parallel

Ke Liu, Sven Loeffler, Petra Hofstedt

Brandenburg University of Technology Cottbus-Senftenberg, Germany

Constraint programming (CP) is a declarative programming paradigm in which the problem is modeled as a group of relations over a finite set of variables. Formally, A *constraint network* $\mathcal{R}$ is a triple (X, D, C), which consists of:

- a finite set of variables $X = \{x_1, \ldots, x_n\}$,
- a set of respective finite domains $D = \{D_1, \ldots, D_n\}$, where D_i is the domain of the variable x_i, and
- a set of constraints $C = \{c_1, ..., c_t\}$, where a constraint c_j is a relation R_j defined on a subset of variables S_j, $S_j \subseteq X$.

Note that from the problem definition point of view, a problem modeled as a constraint network is also called constraint satisfaction problem, or CSP. By using constraint programming, one can easily attack some NP-complete problems in a reasonable execution time without developing specific, and fairly often complex, algorithms.

Backtrack search and constraint propagation lie at the heart of the techniques for constraint programming. But the potential of these techniques might have been exhausted due to the extensive and effective researches in this area during last decades. On the other hand, parallel computing is often an effective approach to enhance the performance of sequential algorithm. However, the research of parallel constraint solving is still in its infant stage although parallel search and parallel propagation for constraint programming have been studied.

This paper aims at presenting a new mapping algorithm as the first step for parallel constraint solving, where the term parallel constraint solving refers to executing the whole constraint network simultaneously by means of partition and mapping it onto the different cores. The idea behind this parallel constraint solving is to divide the original intractable problem into the tractable sub-problems because many NP-complete and NP-hard problems can be solved in polynomial time if the corresponding *hypergraph* has the bounded *hypertree-width* [1]. A hypergraph $\mathcal{H} = (V, S)$ corresponding to a constraint network $\mathcal{N}$ is composed of a set of vertexes $V = \{v_1, ..., v_n\}$ and a set of hyperedges of these vertexes $\mathcal{S} = \{s_1, ..., s_i\}$, where each hyperedge s_i might contain more than two vertexes. Moreover, each hyperedge s_i in a hypergraph $\mathcal{H}$ has a one-to-one correspondence with a constraint in the corresponding constraint network $\mathcal{N}$; similarly, any vertex v_n in the hypergraph $\mathcal{H}$ has a one-to-one correspondence with a variable in $\mathcal{N}$ as well. A *hypertree* of a hypergraph $\mathcal{H}$ is a triple $(T,\ \chi,\ \lambda)$, where $T = (V_T, E_T)$ denotes the tree structure of the hypergraph, χ and λ are labeling functions that stand for the set of variables and the set of constraints of each

node for the hypertree. The hypertree width of a given hypertree is the maximal number of hyperedges (constraints) among all nodes of that hypertree.

Although there exists several hypertree decomposition algorithms, e.g. $det-k-decomp$ [1], in the literature published in the last decades, the target decomposition tree of these algorithms are the hypertree with width as small as possible because the small hypertree width indicates the problem can be solved faster. Hence, we develop a dedicated hypergraph decomposition method $detk-k-decomp$ for mapping constraints onto cores. The idea behind $det-k-CP$ is to utilize the property that one edge between two nodes can be eliminated without changing the set of all solutions for the constraint network [2] because there exists an alternative path between the two nodes. The edge between two nodes represents shared variables. As can be seen from Figure 1, the edge between node c_1 and c_3 can be removed since there is an alternative path $c_1 \rightarrow c_2 \rightarrow c_3$ from c_1 to c_2.

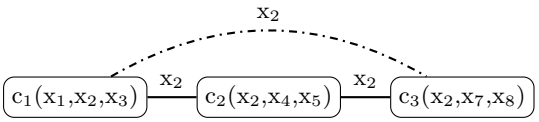


Fig. 1: A simple constraint network shows the edge between two nodes that can be eliminated.

The target decomposition tree of $det-k-CP$ can be represented by a degeneration decomposition tree in which the edges between any pair of non-adjacent nodes are eliminated, and only the edges between adjacent nodes are preserved. A target degeneration decomposition tree with four nodes for a given constraint network is shown in Figure 2 below, where solid lines are drawn for edges between adjacent nodes, and the dot-dashed lines stand for the edges between non-adjacent nodes that can be removed. Besides, the path from $Node1$ to $Node4$, which is drawn by solid line in 2, is also called *main path* of the degeneration decomposition tree.

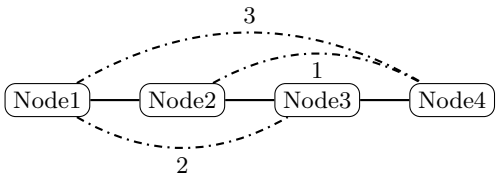
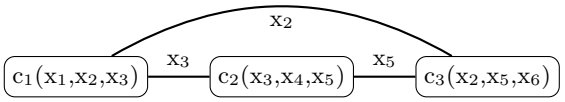
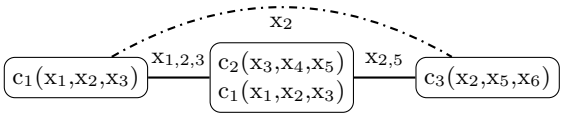


Fig. 2: A degeneration decomposition tree decomposed by $det-k-CP$. The number next to each arc denotes the execution order of adding procedure.



(a) The original constraint network



(b) A simple degeneration decomposition tree

Fig. 3: Adding procedure for eliminating non-adjacent edge

We are now going to sketchily introduce the details of algorithm of $det - k - CP$ by using Figure 2, and Figure 3. In the first step, $det - k - CP$ sorts the constraints based on the weight of each constraint, where the weight of a constraint depends on the time complexity propagation algorithm used by the constraint, the size of domains for variables, and the number of variables for the given constraint network. Then, $det-k-CP$ adds constraints from i_{th} node onto $i+1_{th}$ node if necessary, so that the shared variables between two non-adjacent nodes are covered by the corresponding main path's variables from node j to node k, where $j \leq i < i+1 \leq k$. For example, in Figure 3a, a simple constraint network is composed of three constraints in which the edge between c_1 and c_3 is due to common shared variable x_2. By adding c_1 to the second node, the edge between c_1 and c_3 can be eliminated since the main path contains x_2 now, as shown in Figure 3b. We call this procedure is *adding procedure* for eliminating non-adjacent edge. $det - k - CP$ carries out the adding procedure from the tail to the head of degeneration decomposition tree in turn, e.g., in Figure 2, the edge between $Node2$ and $Node4$ is firstly removed, and then is the edge between $Node1$ and $Node3$. Finally, the edge between $Node1$ and $Node4$ is removed. However, the adding procedure cannot ensure the hypergraph of a given constraint network is decomposed successfully. For instance, in order to eliminate the edge between $Node1$ and $Node3$ in Figure 2, $det - k - CP$ might need to add constraints from $Node1$ to $Node2$ so that the main path between $Node1$ and $Node3$ can cover arc 2. The adding procedure to $Node2$ might regenerate the arc 1 that had been removed by the adding procedure to $Node3$ because the new common shared variables between $Node2$ and $Node4$ might reappear. Therefore, in the last step of $det - k - CP$, the stochastic exchange for nodes is introduced. The heuristic used by stochastic exchange could be randomly exchange, switching nodes that have the fewest and most number of constraints, or changing the permutation of the indexes of degenerate decomposition tree in order. For instance, if the number of nodes of a degenerate decomposition tree is 4, we might firstly use the permutation (0,1,2,3), then (0,1,3,2) and so on. In short, $det - k - CP$ performs adding procedure for every edge between non-adjacent

nodes, and then the stochastic exchange will be executed if the hypergraph of a given constraint network is not decomposed successfully. Adding procedure and stochastic exchange are carried out in each iteration until a degeneration decomposition tree is obtained.

We have briefly presented the decomposition algorithm $det-k-CP$ to map constraints for parallel constraint solving. This algorithm can be solved in polynomial time, because even if there is no degeneration decomposition tree to be found, the algorithm will eventually stop, in that case, every node would contain the entire constraint network. But we have not encountered such a situation so far after evaluating the benchmark suit provided by Gottlob et al. used in study [1]. The future work will be focused on using $det-k-CP$ to gain the speed up for parallel constraint solving.

References

1. Gottlob, G., Samer, M.: A backtracking-based algorithm for hypertree decomposition. Journal of Experimental Algorithmics (JEA) **13** (2009) 1
2. Dechter, R.: Constraint processing. Morgan Kaufmann (2003)

Vectorizing Mathematical Expressions (Extended Abstract)

Joachim Giesen, Julien Klaus, Sören Laue*

Friedrich-Schiller-University Jena, Germany
{joachim.giesen,julien.klaus,soeren.laue}@uni-jena.de

Abstract. Mathematical expressions are ubiquitous in science, engineering and business. They are mostly given in index form, where elements of a data set are addressed by an index. A different, but equivalent representation for mathematical expressions is their vectorized form, where data sets are represented by vectors or matrices. Typically the vectorized representation of an expression can be evaluated much faster, because it can easily be mapped onto highly tuned libraries for basic linear algebra subroutines (BLAS). Evaluating expressions in vectorized form can be a few orders of magnitude faster than evaluating the same expression in index form.
In this paper we present a tool that transforms a mathematical expression in index form, into an equivalent vectorized form. We define a simple grammar for index form expressions, which is parsed into an expression tree. The expression tree is then transformed into another tree that only contains tensors and operations on tensors. Finally the vectorized expression is derived by applying simple substitution rules on the tensor tree. Numerical tests demonstrate the efficiency and correctness of our approach.

Keywords: mathematical expressions, vectorizing, vector unit

1 Introduction

Interpreted languages like Matlab or Python often suffer from slow execution of loops [4,5]. Loops occur naturally in mathematical expressions containing, among others, a universal quantifier or a sum symbol. Two examples of mathematical problems that contain these symbols are logistic regression (LR) or support vector machines (SVM). Both problems are used for classifying labeled data points. Given a data matrix $X \in \mathbb{R}^{m \times n}$ and a label vector $y \in \{-1, +1\}^m$, the goal of both problems is finding a hyperplane that separates the points with label -1 from the points with label $+1$. A separating hyperplane $\{x \in \mathbb{R}^n | x^\top w + b = 0\}$ is represented by $w \in \mathbb{R}^n$ the normal vector to the hyperplane and the offset $b \in \mathbb{R}$. Note that $\frac{|b|}{\|w\|}$ is the distance of the hyperplane from the origin.

* Sören Laue acknowledges the support of Deutsche Forschungsgemeinschaft (DFG) under grant LA-2971/1-1.

The logistic regression problem reads as:

$$\min_{w \in \mathbb{R}^m} \quad \sum_{i=1}^{n} \log\left(1 + \exp\left(\sum_{j=1}^{m} -y_i \cdot w_j \cdot X_{ij} + b\right)\right), \tag{LR}$$

and the support vector machine problem is the following optimization problem:

$$\begin{aligned} \min_{w,\xi} \quad & \frac{1}{2}\sum_{i=1}^{n} w_i^2 + c\sum_{i=1}^{m} \xi_i \\ \text{s.t.} \quad & \forall i : \sum_{j=1}^{m} y_i \cdot (w_j \cdot X_{ij} + b) \geq 1 - \xi_i \\ & \forall i : \xi_i \geq 0. \end{aligned} \tag{SVM}$$

The equivalent representation for the logistic regression in vectorized form is the following:

$$\min_{w \in \mathbb{R}^m} \quad \mathbb{1}^{\mathrm{T}} \cdot \log(\mathbb{1} + \exp(-y \odot (X \cdot w + \mathbb{1} \cdot b))), \tag{LRV}$$

and the support vector machine is given as:

$$\begin{aligned} \min_{w,\xi} \quad & \frac{1}{2} \cdot \|w\|^2 + c \cdot \mathbb{1}^{\mathrm{T}} \cdot \xi \\ \text{s.t.} \quad & \mathbb{1}^{\mathrm{T}} \cdot (y \odot (X \cdot w + \mathbb{1} \cdot b)) \geq \mathbb{1} - \xi \\ & \xi \geq \mathbb{0}, \end{aligned} \tag{SVMV}$$

where $\mathbb{0}$ and $\mathbb{1}$ are the vectors with only zeros and ones, $\|\cdot\|^2$ the Euclidean norm and $\odot$ is the element-wise multiplication. We are interested in a tool, that derives the vectorization automatically.

Usually two nested loops over the indices i and j are used to evaluate the expressions from above. It has been suggested by many authors [2,7] that the problems should be vectorized by hand, since the execution of these loops can be slow.

Vectorization is often nontrivial especially for more complex expressions. Furthermore, many different numerical packages such as Theano [6], Tensorflow [3], or Numexpr [1] require the input to be in vectorized form, i.e., in the form of linear algebra matrix expressions.

In the remainder of this paper we demonstrate our approach for automatically vectorizing mathematical expressions, using logistic regression and support vector machines as examples. After briefly describing the transformation and validation in Section 2, we demonstrate the speedup in evaluating the vectorized expressions experimentally in Section 3, concluding in Section 4.

2 Building Blocks of the Vectorization Tool

Our vectorization tool has three components, namely a parser, a transformation and a validation module that we briefly describe in this section.

2.1 Parsing

For our implementation we have restricted the allowed operations and variables. For the allowed mathematical expressions we have defined a simple grammar, that is shown in Figure 1 using simple EBNF rules.

⟨*formula*⟩ ::= { 'forall' '[' ⟨*index*⟩ {',' ⟨*index*⟩ } '] '} ⟨*compassign*⟩

⟨*compassign*⟩ ::= ⟨*expr*⟩ [(':=' | '==' | '>' | '>=' | '<' | '<=') ⟨*expr*⟩]

⟨*expr*⟩ ::= ⟨*term*⟩ {('+' | '-') ⟨*term*⟩ }

⟨*term*⟩ ::= ⟨*factor*⟩ {('*' | '/')}

⟨*atom*⟩ ::= number | ⟨*function*⟩ '(' ⟨*expr*⟩ ')' | ⟨*variable*⟩

⟨*index*⟩ ::= alpha

⟨*variable*⟩ ::= alpha+ ['[' ⟨*index*⟩ [',' ⟨*index*⟩']']

⟨*function*⟩ ::= 'sin' | 'cos' | 'exp' | 'log' | 'sign' | 'sqrt' | 'abs' | 'sum' '[' ⟨*index*⟩ ']'

Fig. 1. Grammar for mathematical expression.

Using this grammar we can represent the logistic regression problem as follows:

$$\text{sum}[i](\log(1 + \exp(\text{sum}[j](-y[i] * w[j] * X[i,j] + b)))). \tag{1}$$

The first constraint of the support vector machine can be represented as:

$$\text{forall}[i] : \text{sum}[j](y[i] * (w[j] * X[i,j] + b)) >= 1 - xi[i]. \tag{2}$$

After successfully parsing the expression into an expression tree, the transformation into vectorized form follows.

2.2 Transformation

The transformation into vectorized form works in a bottom-up fashion, by iterating through the nodes of the expression tree and replacing the nodes according to their operation. During the transformation the dimensions of the nodes are tracked and adjusted if needed. For example, while transforming the logistic regression expression tree, the algorithm reaches the node + with already transformed subtrees 1 and $\exp(-y. * (X * w + \text{vector}(1) * b))$. Since the right subtree is a vector and the left subtree is a scalar, the algorithm changes the left subtree to $\text{vector}(1) * 1$. After this change the + operator can be processed. Adapting the whole transformation step to tensors reduces the number of dimension checks and simplifies the algorithm. Finally, the resulting vectorization is determined by simple substitutions of tensor algebra expressions by linear matrix algebra.

After a successful transform the algorithm produces the following vectorization for logistic regression example:

$$\text{sum}(\log(\text{vector}(1) + \exp(-y. * (X * w + \text{vector}(1) * b)))), \tag{3}$$

and for the first constraint of the support vector machine:

$$y. * (X * w + \text{vector}(1) * b) >= \text{vector}(1) - xi, \tag{4}$$

where vector(1) is the vector of all ones and $.*$ is the element-wise multiplication.

2.3 Verification

For validating the vectorization against the given formula in index form we perform a numerical check as well as a dimension check. The numerical verification simply calculates values for the unvectorized formula and the vectorized form and checks for equality. The second test compares the dimension of the two formulas. Both must be equal to the number of unbound indices. Only if these tests are correct the vectorization is assumed to be valid.

3 Experiments

For demonstrating the efficiency of our vectorization we have implemented two different versions for each problem. The first version uses loops to implement Equations 1 and 2, whereas the second version directly implements the vectorizations (Equations 3 and 4) using Numpy [8]. From Equations 2 and 4 we only use the left term for the time measurements which is the most time consuming part. Figures 2 and 3 show the evaluation times for the two problems. The run-

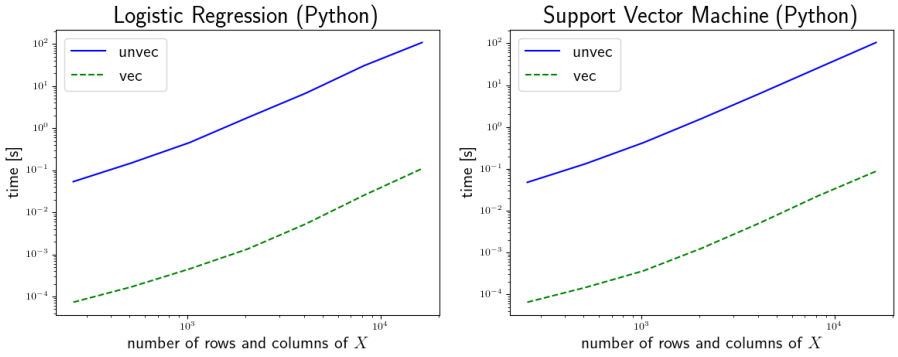


Fig. 2. The comparison of unvectorized formulas against its vectorized form in Python in a logarithmic plot.

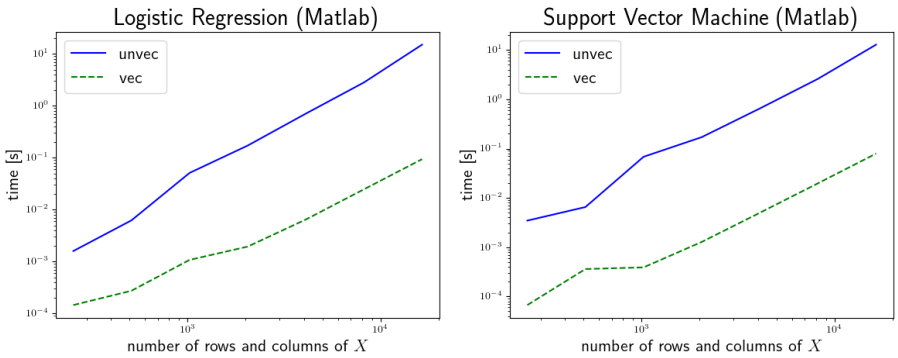


Fig. 3. The comparison of unvectorized formulas against its vectorized form in Matlab in a logarithmic plot.

ning times where obtained with Python 3.6 and Matlab R2016a. We generated random data sets $X \in \mathbb{R}^{m \times n}$ and $y \in \{-1, +1\}^m$ with $m = n$. During the experiments the best of three runs for each problem size is reported. We observe, that the vectorized expression for logistic regression can be evaluated three orders of magnitude faster than the unvectorized expression. Similar results hold for the support vector machine problem.

4 Conclusion

We have presented a tool for automatically transforming some mathematical expressions into their vectorized form. The experiments corroborate the practical benefits of vectorizing mathematical expressions. Especially for large data sets the vectorization mathematical expressions can result in a speedup of a few orders of magnitude. In the future we will include more operators and hence extend the set of mathematical expressions that can be transformed.

The presented tool is available at `http://www.autovec.org`.

References

1. Numexpr: Fast numerical expression evaluator for numpy, `https://github.com/pydata/numexpr`, accessed: 06.07.2017
2. Techniques to improve performance, `https://de.mathworks.com/help/matlab/matlab_prog/techniques-for-improving-performance.html`, accessed: 06.07.2017
3. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V.,

Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., Zheng, X.: TensorFlow: Large-scale machine learning on heterogeneous systems (2015), http://tensorflow.org/, software available from tensorflow.org
4. Cai, X., Langtangen, H.P., Moe, H.: On the performance of the python programming language for serial and parallel scientific computations. Sci. Program. 13(1), 31–56 (Jan 2005), http://dx.doi.org/10.1155/2005/619804
5. David Ascher, Paul F. Dubois, K.H.J.H.T.O.: Numerical python. Tech. rep., Lawrence Livermore National Laboratory, Livermore, CA 94566 (2001)
6. Theano Development Team: Theano: A Python framework for fast computation of mathematical expressions. arXiv e-prints abs/1605.02688 (May 2016), http://arxiv.org/abs/1605.02688
7. Varoquaux, G.: Writing faster numerical code, http://www.scipy-lectures.org/advanced/optimizing/#writing-faster-numerical-code, accessed 06.07.2017
8. van der Walt, S., Colbert, S.C., Varoquaux, G.: The numpy array: A structure for efficient numerical computation. Computing in Science Engineering 13(2), 22–30 (March 2011)

Hyperedge Replacement Grammars for Lock-sensitive Analysis of Parallel Programs

Sebastian Kenter

Institut für Informatik, Westfälische Wilhelms-Universität, Germany
s.k@wwu.de

Abstract. Reachability problems for parallel programs in the presence of locks are often close to the border of decidability. We consider Dynamic Pushdown Networks modelling parallel programs with unboundedly many locks and use Hyperedge Replacement Grammars (HRGs) to analyze the graph structures that represent their executions, exploiting the decidability of the emptiness of a set of graphs generated by an HRG and satisfying certain logical formulas.

Keywords: parallel programs, dynamic pushdown networks, locking, graph grammars

1 Introduction

More and more parallel programs are being used in many areas, and it can be very important to find out if they work properly. Thus we analyze programs with multiple threads and locking, and we are interested in the *lock-sensitive reachability problem* which asks if there is a feasible execution in this setting reaching a state from a given set. Such problems are often close to the border of decidability, and many methods together with restrictions of the general setting have already been developed for solving them. We now analyze programs with unboundedly many locks, which has not been done by prior work in this field. Our approach is to prove decidability by looking at certain graph structures arising from these kinds of programs, and we use Hyperedge Replacement Grammars (HRGs) as a tool to construct those graphs.

2 Program model

The program model we consider in our analysis is based on *Dynamic Pushdown Networks (DPNs)*, which already have been subject to analyses developed in the past (e.g. [1]). A program execution consists of several threads and is run on a state machine with a pushdown for each thread, simulating a call stack for procedures with unbounded recursion depth. New threads can be created dynamically, which leads to unboundedly many threads in this model. Synchronisation between the threads is realized via locking, which is required to abide certain restrictions here:

- Each lock can be *acquired* (once) and *released* later by the same thread.
- As usual, no thread can acquire a lock while it is acquired by another thread.
- At the beginning of an execution, no lock is acquired.
- Whenever a thread acquires another lock l_2 after a lock l_1, it afterwards has to release l_2 before it releases l_1 (*nested locking*).

As a novel extension to this model, called *n-Fold Locking Dynamic Pushdown Network (nLDPN)*, we also allow the dynamic creation of locks, so that the number of locks is unbounded as well. In order to capture the typical situation in common programming languages while keeping the model as simple as possible, we store the locks in a fixed number of procedure-local variables and implement mechanisms to pass them over between procedures.

Formally, an nLDPN is a tuple $M = (V, \mathsf{Act}, P, \Gamma, \Delta, p_0, \gamma_0)$ where

- $V = \{v_1, \ldots, v_n\}$ is a finite set of *variables*,
- $\mathsf{Act} = \mathsf{LockOp} \cup \mathsf{Ass}$ where $\mathsf{LockOp} = \{\operatorname{acq} v, \operatorname{rel} v \mid v \in V\} \cup \{\varepsilon\}, \mathsf{Ass} = \{V \to (V \cup \{\mathsf{new}\})\}$,
- P is a finite set of *control states* with *initial state* $p_0 \in P$,
- Γ is a finite set of *stack symbols* with *initial symbol* $\gamma_0 \in \Gamma$,
- and Δ is a finite set of rules of one of the following forms:

(base)	$p\gamma \overset{lo}{\hookrightarrow} p'\gamma'$
(push)	$p\gamma \overset{a,\omega}{\hookrightarrow} p'\gamma_1\gamma_2$
(pop)	$p\gamma \overset{\varepsilon}{\hookrightarrow} p'$
(spawn)	$p\gamma \overset{a}{\hookrightarrow} p'\gamma' \triangleright p_s\gamma_s$

Here $lo \in \mathsf{LockOp}, a \in \mathsf{Ass}$, and ω is a *return value mapping* with signature $V \to (V \cup \{\mathsf{void}\})$.

This model allows for assigning either the caller's variable values or newly created lock objects (new) to the procedure's local variables on procedure call (push) and either overwriting the caller's variables with return values or or not (void) on procedure return (pop). The locks are acquired and released via the names of the variables that store them using the operations $\operatorname{acq} v$ and $\operatorname{rel} v$, respectively. The semantics, which is not given here in detail, is based on an infinite pool of locks and keeps track of the state, stack content, and variable assignment for each thread, as well as the state of each lock. Because the variables are procedure-local, the assignments are stored on the pushdowns along with the stack symbols so that the caller's values can be accessible again after a procedure returns.

3 Execution Trees

The graph structures that reflect the behaviour of nLDPNs, while abstracting away from the locking mechanism, are called *Execution Trees*. An Execution

Tree is defined such that each vertex represents a program state of a single thread and the paths show all threads' executions that are possible in a given nLDPN. The edges of the tree are labelled with information on the transitions performed by the executions of the threads, taken from the set Δ. Branching in the tree is caused by spawn rules, so that the spawned thread is represented by a different branch, and also by push rules, in which case one branch shows the steps executed up to the corresponding pop operation (i.e., the execution of the "called procedure") and the other one shows everything that happens in the same thread after that (given the called procedure terminates).

Figure 1 shows an example execution of an nLDPN with $P = \{p_0, \ldots, p_9\}$, $\Gamma = \{\gamma, \gamma', \gamma''\}$, and Δ consisting of the following base, push and pop rules:

$$p_0\gamma \hookrightarrow p_1\gamma \qquad p_4\gamma'' \hookrightarrow p_5 \qquad p_7\gamma \overset{\text{rel}\, v_1}{\hookrightarrow} p_8\gamma$$
$$p_1\gamma \overset{a_1}{\hookrightarrow} p_2\gamma'\gamma \qquad p_5\gamma' \overset{\text{acq}\, v_1}{\hookrightarrow} p_6\gamma' \qquad p_8\gamma \overset{\text{acq}\, v_1}{\hookrightarrow} p_9\gamma$$
$$p_3\gamma' \overset{a_2}{\hookrightarrow} p_4\gamma''\gamma' \qquad p_6\gamma' \hookrightarrow p_7 \qquad p_{10}\gamma' \hookrightarrow p_{11}\gamma'$$

and the following spawn rule:

$$p_2\gamma' \overset{a_{sp}}{\hookrightarrow} p_3\gamma' \triangleright p_{10}\gamma'$$

We furthermore assume the initial assignment $v_1 \mapsto l_1$, let $a_1(v_1) = a_2(v_1) = a_{sp}(v_1) = v_1$, and do not consider return value mappings here.

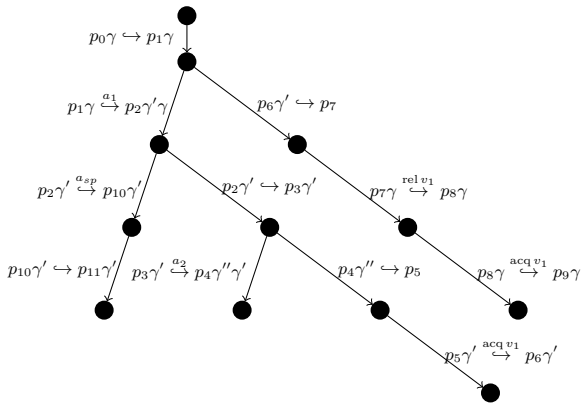


Fig. 1. Example Execution Tree.

The existence of a well-formed Execution Tree showing in its leaves a certain nLDPN configuration (i.e., control states and stack content for each thread) proves that this configuration would be reachable from the considered initial

configuration in the considered nLDPN if there were no locking instructions. However, in presence of locking, one additionally has to find a *lock-sensitive schedule*, i.e. a global ordering of all the steps given in a tree such as to avoid deadlock situations, and those trees where this is impossible (which are not *schedulable*) have to be ruled out. In order to take the this into account, we enrich the trees by the following elements:

- return edges on terminating push branches
- an additional vertex l for each lock $l \in L$
- an additional edge $v \to l$ if l is non-finally acquired at v
- an additional edge $l \to v$ if l is finally acquired at v

By the term *final acquisition* we mean an acquisition of a lock that is never followed by a release of this lock in the considered tree. These enriched structures are called *Augmented Execution Trees (AETs)* (although they are no longer trees). The augmentation arising in our running example is shown in figure 2.

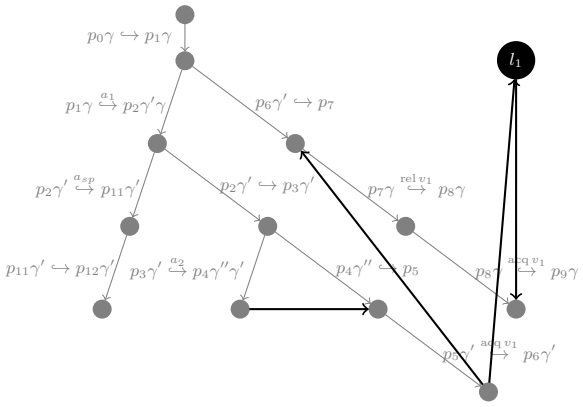


Fig. 2. Augmentation for the example Execution Tree.

In such a graph all edges between program states represent happens-before relations, and with the help of the other vertices and edges one can tell rather plainly if these temporal constraints allow a lock-sensitive schedule of the tree: We can show that an AET has a lock-sensitive schedule if and only if there is not more than one final acquisition of the same lock and the graph is acyclic.

4 A criterion for decidability

Our main statement now turned into the decidability of the existence of an AET satisfying the above-mentioned conditions. We achieve this by identifying certain structural properties of AETs, which are edge-labelled directed graphs,

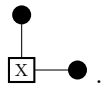
and expressing the additional conditions in *Monadic Second-Order Logic (MSO)* formulas. MSO formulas on graphs view the graphs as logical structures with vertex set V and edge relations $E_a \subseteq V \times V$ for each label a. Quantification is allowed over vertices $v \in V$ as well as subsets $V' \subseteq V$. We can express the relevant constraints, among those the configuration which has to be reached and acyclicity, in this language. Decidability of MSO satisfiability on graphs has been proven for certain cases. Courcelle and Engelfriet state such a result for so-called *HR-equational sets* of graphs ([2], p. 408, theorem 5.80 (2)), which consist of graphs that can be defined in a certain way by specific algebraic graph operations. One possibility to create such a set of graphs is to specify a *Hyperedge Replacement Grammar (HRG)* of graphs.

5 Hyperedge Replacement Grammars

HRGs can be seen as generalizations of context-free grammars of words; they generate HR-equational sets of graphs. We use them also for making sure that the operations given in an AET are consistent to the states and transitions of the nLDPN under consideration.

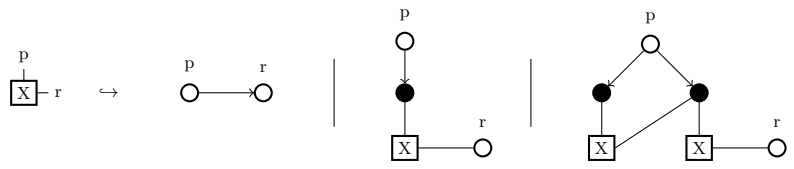
The non-terminals of an HRG are *hyperedges*, i.e. edges connecting an arbitrary number of vertices via destinguished connectors. The production rules contain on their right-hand sides graphs with possibly inserted non-terminals and destinguished *external vertices* matching the connectors on the left-hand side of the rule. The rules specify how a terminal graph can be derived (in finitely many steps, similarly to context-free grammars) from some initial graph that contains non-terminals. In each derivation step, a non-terminal is replaced by the right-hand side of a corresponding rule, where each external vertex in turn is replaced by the vertex that is referenced by the corresponding connector of the non-terminal.

We show an example HRG with one non-terminal called X having two connectors called v and r that generates tree-like graphs with "return edges" connecting the two branches as if the branching vertices were push operations in an AET, out of the initial structure



.

Its three production rules are (in a compact representation):



The rules express the termination of a branch, linear continuation, and branching, respectively. The external vertices are depicted without filling and annotated with the name of the connector whose referenced vertex should replace it in a derivation step. This way, each newly introduced vertex has a "previous vertex" (p) and a "return vertex" (r), which always has to be specified in this example, even if there is no branching.

The transition rules of a given nLDPN can be translated into production rules like these, complemented by edge labels, vertices for the locks and other things, in order to create corresponding AETs.

6 Overview and conclusion

To summarize, we show that the lock-sensitive reachability problem for nLDPNs is decidable by reducing it to MSO satisfiability on HR-relational sets of graphs: We transform the given nLDPN into an HRG generating the corresponding AETs and we specify further conditions by MSO formulas. This establishes a new decidability result for programs with unboundedly many locks in the shape of a flexible and practical program model. We also hope to find decidability theorems for otherwise extended program models using the same method in the future.

References

1. Lammich, P., Müller-Olm, M., Wenner, A.: Predecessor Sets of Dynamic Pushdown Networks with Tree-Regular Constraints. In: Proceedings of Computer Aided Verification (CAV 2009). Springer, Heidelberg (2009). LNCS 5643.
2. Courcelle, B., Engelfriet, J.: Graph Structure and Monadic Second-Order Logic: A Language-Theoretic Approach. Cambridge University Press (2012).

Varianten der modularen Typableitung für Dart

Thomas S. Heinze[1], Anders Møller[2] und Fabio Strocco[2]

[1] Friedrich-Schiller-Universität Jena, Ernst-Abbe-Platz 2, D-07743 Jena, Germany
t.heinze@uni-jena.de
[2] Aarhus Universitet, Åbogade 34, DK-8200 Aarhus N, Denmark
[amoeller,fstrocco]@cs.au.dk

Zusammenfassung. Um auch in dynamisch typisierten Programmiersprachen die Vorteile der statischen Typisierung zu nutzen, unterstützen moderne dynamische Sprachen oft Typannotationen. Zusätzlich kann eine statische Typableitung weitere Abschätzungen zu möglichen Laufzeittypen liefern. Da die Analyse auf die annotierten Typen zurückgreifen kann, bietet sich ein modularer Analyseansatz an, der aber die Sicherheit der Typannotationen berücksichtigen muss. In dieser Arbeit stellen wir Varianten der modularen Typableitung für die Sprache Dart vor.

1 Einführung und Motivation

Dynamische Programmiersprachen können auf vielfältige Weise von der Integration statischer Typen profitieren, sei es bei der Programmoptimierung, bei der Fehlersuche oder in Werkzeugen zur Programmentwicklung, wie der Codenavigation und -vervollständigung. Für ein dynamisch typisiertes Programm lassen sich Typen in Form von Typannotationen angeben, durch eine statische Typableitung abschätzen, oder mittels einer Kombination aus beidem bestimmen. Letzterer Ansatz ist in Verbindung mit Konzepten der optionalen oder graduellen Typisierung in modernen Sprachen wie *TypeScript*, *Python*, *Dart*, bei denen statisch als auch dynamisch typisierte Abschnitte gleichzeitig in einem Programm vorliegen können, vielversprechend. Die in einem Programm annotierten Typen lassen sich dabei zur Umsetzung einer *modularen Typableitung* ausnutzen.

In [4,5] haben wir zur Unterstützung der statischen Typprüfung von Dart-Programmen bereits Möglichkeiten der Typableitung vorgestellt. Ein Programm der objektorientierten, klassenbasierten und optional typisierten Sprache *Dart* [3] ist grundsätzlich dynamisch typisiert, das heißt verwendet implizit den dynamischen Typ `dynamic`. Allerdings können dem Programm auch Typen annotiert werden, mit denen entsprechende Typprüfungen zur Programmlaufzeit definiert werden.[1] In der Folge können für ein Dart-Programm zwei typbezogene Laufzeitfehler auftreten: Einerseits wird ein Fehler beim Zugriff auf eine undefinierte

[1] Dart unterstützt zwei Ausführungsmodi: Ein Programm kann in Übereinstimmung mit der optionalen Typisierung entweder vollständig dynamisch und unabhängig von den annotierten Typen ausgeführt werden (*Production Mode*), oder es erfolgen Typprüfungen zur Programmlaufzeit anhand der Typannotationen vergleichbar der graduellen Typisierung (*Checked Mode*). Wir beziehen uns hier auf letzteren Modus.

```
1: class Box<E> { E v; }
2: void main() {
3:   var x = new Box<dynamic>();
4:   x.v = new Object();
5:   Box<int> y = x;
6:   bool i = y.v.isEven;
7: }
```

```
1: class Sup { String f; }
2: class Sub extends Sup { Object f; }
3: void main() {
4:   Sub x = new Sub();
5:   x.f = new Object();
6:   Sup y = x;
7:   String s = y.f;
8: }
```

Abb. 1. Programmbeispiele für unsichere Typen in Dart

Methode beziehungsweise ein undefiniertes Feld ausgelöst, andererseits bei Vorliegen einer Unverträglichkeit von Laufzeittyp und annotiertem Typ.

Ausgangspunkt von [4,5] war, die statische Typprüfung hinsichtlich des Auftretens dieser zwei Laufzeitfehler zu unterstützen, indem insbesondere für die dynamisch typisierten Programmabschnitte sichere Abschätzungen zu möglichen Laufzeittypen abgeleitet werden. Grundsätzlich sind für eine solche Typableitung zwei Ansätze denkbar, wie in [5] unter den Begriffen *Flow Mode* und *Modular Mode* diskutiert. Zum einen kann der vollständige Fluss von Laufzeittypen im Programm nachvollzogen werden. Die Typableitung lässt sich dann in Form einer ANDERSEN-Analyse [12] realisieren. Zum anderen kann die Analyse aber die annotierten Typen als Spezifikationen auffassen und modular, das heißt ohne vollständige Analyse des Datenflusses im Programm, ableiten.

Gegenüber der Typabschätzung auf Grundlage einer vollständigen Analyse des Datenflusses, bietet ein solcher modularer Ansatz verschiedene Vorteile, insbesondere hinsichtlich Robustheit und Skalierbarkeit der Analyse [5]. Anders als für statisch typisierte Sprachen, ist eine sichere Typabschätzung mittels modularem Analyseansatz jedoch im Fall der dynamischen Typisierung nicht ohne Weiteres möglich. Für generische Typen stellt sich etwa das Problem, dass sich deren Typgarantien unter Verwendung von `dynamic` umgehen lassen. In Abbildung 1 ist auf der linken Seite ein entsprechendes Dart-Programm dargestellt. Die Zuweisung von `Box<dynamic>` an `y` ist aufgrund der Einordnung des dynamischen Typs `dynamic` in der Typhierarchie zulässig, so dass es zur Programmlaufzeit erst beim Zugriff auf das Feld `isEven` in Zeile 6 zu einem Fehler kommt, da der Laufzeittyp `Object` von `y.v` dieses nicht definiert. Bestehende Werkzeuge zur statischen Typprüfung für Dart (*dartanalyzer*) erkennen diesen Fehler nicht.[2]

In dieser Arbeit stellen wir überblicksweise drei verschiedene Ansätze zur modularen Typableitung vor: Der erste Ansatz vernachlässigt das Problem unsicherer Typen und entspricht einer naiven Übertragung des Prinzips der statischen Typprüfung für Programmiersprachen wie Java. Der zweite Ansatz wählt einen *pessimistischen* Zugang und geht von der Unsicherheit annotierter Typen aus. Der dritte Ansatz setzt schließlich ein *optimistisches* Verfahren unter Annahme sicherer Typen um, beinhaltet aber gleichzeitig eine vollständige Analyse des Datenflusses für die während der Analyse identifizierten unsicheren Typen.

[2] Mit Ausnahme des sogenannten *Strong Mode*, siehe dessen Diskussion in Abschnitt 4.

T ::= $N \mid E \mid$ `dynamic`
N ::= C`<`$\overline{T}$`>`
L ::= `class` C`<`$\overline{E}$`>` `extends` N `{` $\overline{T}\ \overline{f}$`;` `}`
e ::= `new` N`()` $\mid e.f \mid x \mid x = e \mid e.f = e \mid$ `(`N`)` e

x ... Variablenbezeichner
f ... Feldbezeichner
C, D ... Klassennamen
E ... Typparameter

Abb. 2. Betrachtetes Sprachfragment von Dart über Feld- und Variablenzugriffen

2 Naive modulare Typableitung für Dart

Zur Veranschaulichung des modularen Analyseansatzes wird das in Abbildung 2 dargestellte Sprachfragment, in Anlehnung an *Featherweight Java* [6], verwendet. Der Einfachheit halber beschränken wir uns auf Feld- und Variablenzugriffe, so dass das abgebildete Fragment neben Klassen- und Felddeklarationen lediglich Ausdrücke über Objektinstanziierungen, explizite Typumwandlungen, sowie lesenden und schreibenden Feld- beziehungsweise Variablenzugriff umfasst.[3] Dabei bezeichne $\overline{x}$ die eventuell leere Sequenz $x_1 \ldots x_n, n \geq 0$. Variablen sind grundsätzlich dynamisch typisiert, Typannotationen können aber über die Kombination aus Variablenzugriff und Typumwandlung simuliert werden, so etwa mittels `y = (Box<int>) x;` für die Zuweisung in Zeile 5 aus Abbildung 1.

Eine *modulare Typableitung* kann die Typannotationen nutzen, so dass etwa zur Abschätzung des Laufzeittyps für den Teilausdruck `y.v` aus Abbildung 1 auf den der Variable `y` annotierten Typ `Box<int>` zurückgegriffen und dementsprechend `int` abgeleitet wird. Für ein vollständig mit statischen Typen annotiertes Programm würde dieser Ansatz somit der Typprüfung für rein statisch typisierte Sprachen entsprechen. Sind in einem Programm dynamisch typisierte, das heißt mit `dynamic` ausgezeichnete, Abschnitte vorhanden, werden die im Programm instanziierten und annotierten Typen zusätzlich entlang des relevanten Datenflusses propagiert. Dieser grundlegende Ansatz lässt sich im Rahmen einer Andersen-Analyse [12], wie in Abbildung 3 angegeben, formalisieren.

Dazu werden den Ausdrücken und Feldern eines Programms Typvariablen zugewiesen, denen Mengen von Typen zugeordnet sind (Funktion $[\![]\!]$ in Abbildung 3). Wie zu erkennen, wird für abgeleitete Typen τ, σ, ρ dabei zwischen *konkreten Typen* N^C und *abstrakten Typen* N^A unterschieden, wobei letztere auch ihre jeweiligen Untertypen umfassen (vergleiche Relation $\trianglelefteq$ in Abbildung 3). Weiterhin werden Regeln zwischen den Typvariablen in Form von Mengenbeziehungen definiert. Als Lösung des dadurch charakterisierten Regelsystems ergibt sich dann die Abschätzung zu den möglichen Laufzeittypen des Programms.

Durch die Regeln wird prinzipiell zwischen mit Typen annotierten und dynamisch typisierten Variablen und Feldern unterschieden. Beim Zugriff auf erstere wird der Fluss der Typen nachvollzogen, für letztere wird hingegen der annotierte Typ abgeleitet. Beim Zugriff auf ein dynamisch typisiertes Feld f für einen abstrakten Typ τ (Prädikat $fdyn(\tau, f)$) ist dabei zu beachten, dass auch dessen Untertypen berücksichtigt werden, da diese das Feld überschreiben können.[4]

[3] Wir schließen zur Vereinfachung (ungebundene) Typparameter in Ausdrücken aus.
[4] Im Gegensatz zu Java werden Felder in der Programmiersprache Dart überschrieben.

$$\tau, \sigma, \rho ::= N^C \mid N^A \quad N^C \trianglelefteq N^C \quad N^C \trianglelefteq N^A \quad \frac{C\texttt{<}\overline{T}\texttt{>} <: D\texttt{<}\overline{T}\texttt{>}}{C\texttt{<}\overline{T}\texttt{>}^C \trianglelefteq D\texttt{<}\overline{T}\texttt{>}^A} \quad \frac{\mathit{ftype}(\tau, f) = \texttt{dynamic}}{\mathit{fdyn}(\tau, f)}$$

Instanziierung $\texttt{new}\ N()$:
$$N^C \in [\![\texttt{new}\ N()]\!]$$

Zuweisung $x = e$:
$$[\![e]\!] \subseteq [\![x]\!]$$

Typumwandlung $(N)\ e$:
$$N^A \in [\![(N)\ e]\!]$$

Feldzugriff $e.f$:
$$\frac{\tau \in [\![e]\!] \quad \mathit{ftype}(\tau, f) = N}{N^A \in [\![e.f]\!]}$$
$$\frac{\tau \in [\![e]\!] \quad \mathit{fdyn}(\tau, f) \quad \sigma \trianglelefteq \tau \quad \mathit{ftype}(\sigma, f) = N}{N^A \in [\![e.f]\!]}$$
$$\frac{\tau \in [\![e]\!] \quad \mathit{fdyn}(\tau, f) \quad \sigma \trianglelefteq \tau \quad \mathit{fdyn}(\sigma, f) \quad \sigma = C\texttt{<}\overline{T}\texttt{>}^C}{[\![C.f]\!] \subseteq [\![e.f]\!]}$$

Feldzugriff $e.f = e'$:
$$[\![e']\!] \subseteq [\![e.f\ =\ e']\!]$$
$$\frac{\tau \in [\![e]\!] \quad \mathit{ftype}(\tau, f) = N}{N^A \in [\![C.f]\!]}$$
$$\frac{\tau \in [\![e]\!] \quad \mathit{fdyn}(\tau, f) \quad \sigma \trianglelefteq \tau \quad \mathit{ftype}(\sigma, f) = N \quad \sigma = C\texttt{<}\overline{T}\texttt{>}^C}{N^A \in [\![C.f]\!]}$$
$$\frac{\tau \in [\![e]\!] \quad \mathit{fdyn}(\tau, f) \quad \sigma \trianglelefteq \tau \quad \mathit{fdyn}(\sigma, f) \quad \sigma = C\texttt{<}\overline{T}\texttt{>}^C}{[\![e']\!] \subseteq [\![C.f]\!]}$$

Abb. 3. Modulare Typableitung unter Missachtung unsicherer Typen (*ftype* ermittelt für Felder den annotierten Typ, siehe Anhang A; <: steht für die Subtyprelation [3])

Wird wieder das Beispielprogramm auf der linken Seite von Abbildung 1 betrachtet, so ergibt sich etwa für den Feldzugriff `y.v` anhand der Regeln:[5]

$$\dfrac{\dfrac{\overline{\texttt{Box<int>}^A \in [\![\texttt{(Box<int>) x}]\!]}^{\text{Z5}} \quad \overline{[\![\texttt{(Box<int>) x}]\!] \subseteq [\![\texttt{y}]\!]}^{\text{Z5}}}{\texttt{Box<int>}^A \in [\![\texttt{y}]\!]}^{\text{Z6}} \quad \mathit{ftype}(\texttt{Box<int>}^A, \texttt{v}) = \texttt{int}}{\texttt{int}^A \in [\![\texttt{y.v}]\!]}$$

Werden für den Moment unsichere Typen außer Acht gelassen, liegen die Vorteile dieses modularen Analyseansatzes auf der Hand. Einerseits unterstützen die annotierten Typen die Skalierbarkeit der Typableitung, da nicht der vollständige Fluss der Typen entlang des Datenflusses nachvollzogen werden muss und sich somit die im schlechtesten Fall kubische Analyse [12], in Abhängigkeit vom Vorhandensein der Typannotationen, auf eine lineare Analyse reduziert. Andererseits sind die Analyseergebnisse für Programmkomponenten robuster gegenüber einem rein flussbasierten Ansatz, da sich Änderungen nur bis an die mit Typannotationen versehenen Komponentenschnittstellen auswirken.

Dem aufmerksamen Leser wird nicht entgangen sein, dass der naive Ansatz nicht zu einer sicheren Typabschätzung führt. Für das Programm auf der linken

[5] Regelanwendungen sind mit zugehörigen Zeilennummern im Programm angegeben.

Instanziierung $\texttt{new}\ N()$: $$N^C \in [\![\texttt{new}\, N()]\!]$$

Zuweisung $x = e$: $$[\![e]\!] \subseteq [\![x]\!]$$

Typumwandlung $(N)\ e$: $$\frac{N = C\texttt{<}\overline{T}\texttt{>}}{C\texttt{<}\overline{\texttt{dynamic}}\texttt{>}^A \in [\![(N)\ e]\!]}$$

Feldzugriff $e.f$: $$\frac{\tau \in [\![e]\!] \quad \sigma \trianglelefteq \tau \quad \mathit{ftype}(\sigma, f) = D\texttt{<}\overline{T}\texttt{>}}{D\texttt{<}\overline{\texttt{dynamic}}\texttt{>}^A \in [\![e.f]\!]}$$

$$\frac{\tau \in [\![e]\!] \quad \sigma \trianglelefteq \tau \quad \mathit{fdyn}(\sigma, f) \quad \sigma = C\texttt{<}\overline{T}\texttt{>}^C}{[\![C.f]\!] \subseteq [\![e.f]\!]}$$

Feldzugriff $e.f = e'$: $$[\![e']\!] \subseteq [\![e.f\ =\ e']\!]$$

$$\frac{\tau \in [\![e]\!] \quad \sigma \trianglelefteq \tau \quad \mathit{ftype}(\sigma, f) = D\texttt{<}\overline{T}\texttt{>} \quad \sigma = C\texttt{<}\overline{S}\texttt{>}^C}{D\texttt{<}\overline{\texttt{dynamic}}\texttt{>}^A \in [\![C.f]\!]}$$

$$\frac{\tau \in [\![e]\!] \quad \sigma \trianglelefteq \tau \quad \mathit{fdyn}(\sigma, f) \quad \sigma = C\texttt{<}\overline{T}\texttt{>}^C}{[\![e']\!] \subseteq [\![C.f]\!]}$$

Abb. 4. Pessimistischer Analyseansatz

Seite von Abbildung 1 ist – bedingt durch die Verwendung des Typs `dynamic` – das Aliasing in Zeile 5 zulässig und führt zu keiner Typverletzung. Dadurch werden die mittels Annotation `Box<int>` definierten Typgarantien für die Variable `y` umgangen, so dass für eine sichere Typabschätzung `int` nicht allein als Typ für den Feldzugriff `y.v` abgeleitet werden darf. Tatsächlich enthält das Feld zur Laufzeit ein `Object`, womit es, wie bereits erwähnt, dann in Zeile 6 zu einem Laufzeitfehler kommt. Neben generischen Typen liegt das Problem der unsicheren, das heißt nicht durch Laufzeitprüfungen geschützten Typannotationen in Dart auch für Funktionstypen und Schranken von Typparametern, sowie für das Überschreiben von Feldern und Methoden vor [3]. Für letzteres, vergleiche auch das zulässige Überschreiben von `f` im Dart-Programm auf der rechten Seite von Abbildung 1, für das es zur Laufzeit erst in Zeile 7 zu einem Typfehler kommt.

3 Pessimistischer und optimistischer Ansatz

Um eine sichere Typabschätzung auch für unsichere Typen zu ermöglichen, definieren wir einen *pessimistischen Analyseansatz* anhand der modifizierten Regeln in Abbildung 4. Wie zu erkennen, beziehen wir bei Feldzugriffen für abstrakte Typen nun prinzipiell deren Untertypen mit ein, nicht nur bei dynamisch typisierten Feldern. Für den Feldzugriff `y.f` im Beispiel auf der rechten Seite von Abbildung 1 würde sich somit anhand der modifizierten Regeln ergeben:

$$\dfrac{\dfrac{\dfrac{}{\texttt{Sup}^A \in [\![\texttt{(Sup) x}]\!]}\ \text{Z6} \quad \dfrac{}{[\![\texttt{(Sup) x}]\!] \subseteq [\![\texttt{y}]\!]}\ \text{Z6}}{\text{Z7}\ \ \texttt{Sup}^A \in [\![\texttt{y}]\!]} \quad \dfrac{\texttt{Sub} <: \texttt{Sup}}{\texttt{Sub}^C \trianglelefteq \texttt{Sup}^A} \quad \mathit{ftype}(\texttt{Sub}^C, \texttt{f}) = \texttt{Object}}{\texttt{Object}^A \in [\![\texttt{y.f}]\!]}$$

Instanziierung $\texttt{new}\ N()$:
$$N^C \in [\![\texttt{new}\ N()]\!]$$

Zuweisung $x = e$:
$$[\![e]\!] \subseteq [\![x]\!]$$

Typumwandlung $(N)\ e$:
$$N^A \in [\![(N)\ e]\!]$$
$$\frac{\tau \in [\![e]\!] \quad \tau \trianglelefteq N^A \quad \mathit{unsafe}(\tau)}{\tau \in [\![(N)\ e]\!]} \quad (*)$$

Feldzugriff $e.f$:
$$\frac{\tau \in [\![e]\!] \quad \mathit{ftype}(\tau, f) = N}{N^A \in [\![e.f]\!]}$$
$$\frac{\tau \in [\![e]\!] \quad \mathit{fdyn}(\tau, f) \quad \sigma \trianglelefteq \tau \quad \mathit{ftype}(\sigma, f) = N}{N^A \in [\![e.f]\!]}$$
$$\frac{\tau \in [\![e]\!] \quad \mathit{fdyn}(\tau, f) \quad \sigma \trianglelefteq \tau \quad \mathit{fdyn}(\sigma, f) \quad \sigma = C\texttt{<}\overline{T}\texttt{>}^C}{[\![C.f]\!] \subseteq [\![e.f]\!]}$$
$$\frac{\tau \in [\![e]\!] \quad \sigma \trianglelefteq \tau \quad \sigma = C\texttt{<}\overline{T}\texttt{>}^C \quad \rho \in [\![C.f]\!] \quad \rho \trianglelefteq \mathit{ftype}(\sigma, f)^A \quad \mathit{unsafe}(\rho)}{\rho \subseteq [\![e.f]\!]} \quad (*)$$

Feldzugriff $e.f = e'$:
$$[\![e']\!] \subseteq [\![e.f\ =\ e']\!]$$
$$\frac{\tau \in [\![e]\!] \quad \mathit{ftype}(\tau, f) = N}{N^A \in [\![C.f]\!]}$$
$$\frac{\tau \in [\![e]\!] \quad \mathit{fdyn}(\tau, f) \quad \sigma \trianglelefteq \tau \quad \mathit{ftype}(\sigma, f) = N \quad \sigma = C\texttt{<}\overline{T}\texttt{>}^C}{N^A \in [\![C.f]\!]}$$
$$\frac{\tau \in [\![e]\!] \quad \mathit{fdyn}(\tau, f) \quad \sigma \trianglelefteq \tau \quad \mathit{fdyn}(\sigma, f) \quad \sigma = C\texttt{<}\overline{T}\texttt{>}^C}{[\![e']\!] \subseteq [\![C.f]\!]}$$
$$\frac{\tau \in [\![e]\!] \quad \sigma \trianglelefteq \tau \quad \sigma = C\texttt{<}\overline{T}\texttt{>}^C \quad \rho \in [\![e']\!] \quad \rho \trianglelefteq \mathit{ftype}(\sigma, f)^A \quad \mathit{unsafe}(\rho)}{\rho \subseteq [\![C.f]\!]} \quad (*)$$

Abb. 5. Optimistischer Ansatz (*unsafe* bestimmt unsichere Typen, siehe Anhang A)

so dass der Laufzeitfehler in Zeile 7 erkannt werden kann. Im naiven Ansatz wird stattdessen nur $\texttt{String}^A \in [\![\texttt{y.f}]\!]$ abgeleitet und dieser Fehler übersehen.

Gleichzeitig werden in den Regeln nun die Typargumente von annotierten generischen Typen grundsätzlich ignoriert, das heißt durch `dynamic` ersetzt, unabhängig davon ob der Laufzeittyp tatsächlich `dynamic` als Typargument definiert oder nicht. Auf diese Weise wird beim Zugriff auf ein generisches Feld nicht das Typargument als abstrakter Typ abgeleitet sondern stattdessen der Fluss der Typen in und aus dem Feld nachvollzogen. Für das Beispielprogramm auf der linken Seite von Abbildung 1 ergibt sich im Fall von `y.v` nun $\texttt{Box<dynamic>}^A \in [\![\texttt{y}]\!]$ sowie $\texttt{Object}^C \in [\![\texttt{Box.v}]\!]$ und damit schließlich auch $\texttt{Object}^C \in [\![\texttt{y.v}]\!]$.

Der pessimistische Analyseansatz führt derart zwar zu einer sicheren Typabschätzung, birgt aber Nachteile für Präzision und Skalierbarkeit. So werden die Typgarantien von unsicheren Typen grundsätzlich nicht für die Typableitung ausgenutzt. Dieser Verzicht führt aber gerade wieder zur vollständigen Analyse des Datenflusses, im Widerspruch zum modularen Analyseansatz. Der *optimistische Ansatz* zur modularen Typableitung versucht diese Nachteile zu vermeiden und geht zunächst von der Annahme sicherer Typen aus. Dadurch begründet

sich auch die Ähnlichkeit der zugehörigen Regeln in Abbildung 5 mit denen des naiven Ansatzes aus Abbildung 3. Tatsächlich entsprechen fast alle Regeln einander, einziger Unterschied sind die mit $(*)$ gekennzeichneten zusätzlichen Regeln. Aufgabe dieser ist es, bei Auftreten unsicherer Typen, bestimmt durch das Prädikat *unsafe*, die Analyse auf das Nachvollziehen des Datenflusses umzuschalten. Dazu wird das Prädikat als eine Art Filter verwendet, um nur die unsicheren Typen entlang des Datenflusses zu propagieren, ähnlich der Verwendung von Typfiltern [12]. Tritt kein unsicherer Typ während der Analyse auf, werden wie im naiven Analyseansatz – bei Vorliegen entsprechender Typannotationen – ausschließlich die annotierten Typen abgeleitet. Für das Beispielprogramm auf der linken Seite von Abbildung 1 ergibt sich somit zunächst ebenfalls $\texttt{Box<int>}^A \in [\![\texttt{y}]\!]$ (vergleiche die entsprechende Ableitung in Abschnitt 2). Zugleich wird nun aber auch $\texttt{Box<dynamic>}^C \in [\![\texttt{y}]\!]$ abgeleitet:

$$\dfrac{\dfrac{\dfrac{\texttt{Box<dynamic>}^C \in [\![\texttt{new Box<dynamic>()}]\!] \qquad [\![\texttt{new Box<dynamic>()}]\!] \subseteq [\![\texttt{x}]\!]}{\texttt{Box<dynamic>}^C \in [\![\texttt{x}]\!]}\,\text{Z3} \qquad \dfrac{\dfrac{\texttt{Box<dynamic>} <: \texttt{Box<int>}}{\texttt{Box<dynamic>}^C \trianglelefteq \texttt{Box<int>}^A} \qquad \mathit{unsafe}(\texttt{Box<dynamic>}^C)}{\texttt{Box<dynamic>}^C \in [\![\texttt{(Box<int>) x}]\!]}\,\text{Z5}}{\texttt{Box<dynamic>}^C \in [\![\texttt{y}]\!]}\,\text{Z5} \qquad \dfrac{}{[\![\texttt{(Box<int>) x}]\!] \subseteq [\![\texttt{y}]\!]}\,\text{Z5}$$

und in der Folge ergeben sich für den Feldzugriff `y.v` die Typen `int` und `Object`.

Unter unsicheren Typen werden dabei Typen mit dynamischen Typargument verstanden, wie `Box<dynamic>` im Beispiel, als auch Typen, in denen Felder kontra- oder kovariant überschrieben werden. Der optimistische Ansatz kann derart eine sichere Typabschätzung gewährleisten, falls die in Dart zulässige Kovarianz generischer Typen unberücksichtigt bleibt. Zugleich profitiert der optimistische Analyseansatz von Typannotationen, ähnlich der Typprüfung in rein statisch und stark typisierten Sprachen, da annotierte sichere Typen eben ohne Nachvollziehen des Datenflusses abgeleitet werden können. Offen bleibt für den optimistischen Ansatz jedoch das Problem der Kovarianz generischer Typen, wie beispielsweise im Fall von `Box<Object> z = new Box<int>(); z.f = 1;` da sich dieses nicht allein lokal und effizient auf Typebene entscheiden lässt.[6]

4 Verwandte Arbeiten

Die Programmiersprache Dart bietet einen interessanten Ansatz zur Kombination von statischer und dynamischer Typisierung. Einerseits ist Dart im sogenannten *Production Mode* optional typisiert, in Übereinstimmung mit Gilad Brachas Konzept *Pluggable Type System* [2], wird also unabhängig von annotierten Typen ausgeführt. Andererseits definieren die Typannotationen im sogenannten *Checked Mode* Laufzeittypprüfungen, wodurch sich Dart in gewisser

[6] Denkbar wäre Kovarianz generischer Typen auszuschließen (analog *Strong Mode*).

Weise in das Spektrum der graduellen Typisierung [9,11] einreiht. Im Gegensatz zu dieser werden aber in Dart sowohl bei der statischen Typprüfung (*dartanalyzer*) als auch zur Programmlaufzeit keinerlei Garantien zur Typsicherheit von annotiertem Code gegeben. Wir beziehen uns hier auf *Checked Mode*.

Dart erfährt eine fortlaufende Weiterentwicklung. Wichtig ist in diesem Zusammenhang die geplante Umstellung auf den sogenannten *Strong Mode*[7]. Dieser definiert eine sichere Teilmenge von Dart, ergänzt durch Sprachneuerungen wie generische Methoden und Typableitung, mit dem Ziel ein sicheres Typsystem zu realisieren. Dadurch positioniert sich *Strong Mode* viel näher an der graduellen Typisierung als *Checked Mode*. Nicht mehr unterstützt werden unter anderem das ko-/kontravariante Überschreiben von Feldern, Kovarianz generischer Typen oder auch die Zuweisung von `Box<dynamic>` an die mit `Box<int>` annotierte Variable `y` aus Abbildung 1. Die Typableitung kann somit einfacher erfolgen, vergleichbar dem in Abschnitt 2 vorgestellten naiven Ansatz. Gleichzeitig geht aber Sprachflexibilität verloren, die unsere Analysen stattdessen unterstützt.

Statische Typableitung zur Unterstützung graduell typisierter Sprachen wird in einer Reihe von Forschungsarbeiten behandelt, unter anderem in [1,7,8,10]. Eine frühe Arbeit dazu [10], betrachtet die Kombination von gradueller Typisierung und Typableitung nach Hindley-Milner für funktionale Sprachen, ist aber für objektorientierte Sprachen wie Dart nicht geeignet [7]. In [7] wird eine Typableitung für *ActionScript* beschrieben. Im Gegensatz zu unseren Ansatz, werden keine Mengen von Laufzeittypen zur statischen Typprüfung abgeleitet, sondern Typannotationen für dynamisch typisierte Programmabschnitte, um die Anzahl an Typprüfungen zur Laufzeit zu verringern. Bestehende Typannotationen (sogenannte *Outflows*) bleiben dabei interessanterweise unberücksichtigt. *TypeScript* fügt Typen zu *JavaScript* hinzu und unterstützt den graduellen Typisierungsansatz ebenfalls mit einer statischen Typableitung und -prüfung. Die Analyse verfolgt einen ähnlichen modularen Ansatz wie wir, ist aber aufgrund des Typsystems von *TypeScript* unsicher [1], das heißt kann nicht alle Typfehler statisch finden. Mittels zusätzlicher Laufzeitprüfungen und einem modifizierten Typsystem wird in [8] eine sichere Variante von *TypeScript* beschrieben.

5 Zusammenfassung und Ausblick

Im vorliegenden Beitrag haben wir drei Ansätze zur modularen Typableitung für die optional typisierte Sprache Dart diskutiert. Eine naive Übertragung des Prinzips der statischen Typprüfung, der erste Ansatz, hat sich aufgrund des unsicheren Typsystems von Dart als ungeeignet herausgestellt. Der zweite pessimistische Ansatz führt zwar zu einer sicheren Typabschätzung, bedingt aber gleichzeitig eine Verminderung von Analysepräzision und Skalierbarkeit. Um diese Nachteile zu vermeiden, geht der dritte, optimistische Ansatz hingegen prinzipiell von sicheren Typen aus, beinhaltet zugleich aber die vollständige Analyse des Datenflusses für Verletzungen dieser Annahme beim Überschreiben von Feldern und bei der Verwendung von `dynamic` als Typargument generischer Typen.

[7] `https://www.dartlang.org/guides/language/sound-dart`, Zugriff am 20.8.2017

Offen und zukünftigen Arbeiten überlassen bleibt, für den optimistischen Ansatz das Problem der Kovarianz generischer Typen. Eine genaue Evaluation der drei Analyseansätze hinsichtlich deren Skalierbarkeit wird einen weiteren Schwerpunkt von Arbeiten bilden, wobei uns insbesondere der Zusammenhang zwischen Analyseaufwand und den Eigenschaften des Typsystems von Dart interessiert.

Literatur

[1] Bierman, Gavin ; Abadi, Martín ; Torgersen, Mads: Understanding TypeScript. In: *ECOOP 2014, Object-Oriented Programming, 28th European Conference, Uppsala, Sweden, July 28–August 1, 2014, Proceedings*, Springer, 2014, S. 257–281

[2] Bracha, Gilad: *Plugable Type Systems.* OOPSLA'04 Workshop on Revival of Dynamic Languages. `http://bracha.org/pluggableTypesPosition.pdf`

[3] *Dart Programming Language Specification, 4th Edition.* Standard ECMA-408, 2015

[4] Heinze, Thomas S. ; Møller, Anders ; Strocco, Fabio: Statische Typableitung für die optional typisierte Sprache Dart. In: *18. Kolloquium Programmiersprachen und Grundlagen der Programmierung, Bericht 2015-IX-1, Pörtschach am Wörthersee, 5.–7. Oktober 2015*, Technische Universität Wien, 2015, S. 260–265

[5] Heinze, Thomas S. ; Møller, Anders ; Strocco, Fabio: Type Safety Analysis for Dart. In: *DLS'16, Proceedings of the 12th Symposium on Dynamic Languages, Amsterdam, The Netherlands, November 1, 2016*, ACM, 2016, S. 1–12

[6] Igarashi, Atsushi ; Pierce, Benjamin ; Wadler, Philip: Featherweight Java: A Minimal Core Calculus for Java and GJ. In: *OOPSLA'99, Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, Denver, USA, November 1–5, 1999*, ACM, 1999, S. 132–146

[7] Rastogi, Aseem ; Chaudhuri, Avik ; Hosmer, Basil: The Ins and Outs of Gradual Type Inference. In: *POPL'12, Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Philadelphia, Pennsylvania, USA, January 22–28, 2012*, ACM, 2012, S. 481–494

[8] Rastogi, Aseem ; Swamy, Nikhil ; Fournet, Cédric ; Bierman, Gavin ; Vekris, Panagiotis: Safe & Efficient Gradual Typing for TypeScript. In: *POPL'15, Proceedings of the 42nd Annual ACM Symposium on Principles of Programming Languages, Mumbai, India, January 15–17, 2015*, ACM, 2015, S. 167–180

[9] Siek, Jeremy G. ; Taha, Walid: Gradual Typing for Functional Languages. In: *Proceedings of the 2006 Scheme and Functional Programming Workshop*, University of Chicago, 2006 (Technischer Bericht TR-2006-06), S. 81–92

[10] Siek, Jeremy G. ; Vachharajani, Manish: Gradual Typing with Unification-based Inference. In: *DLS'08, Proceedings of the 2008 Symposium on Dynamic Languages, Paphos, Cyprus, July 8, 2008*, ACM, 2008

[11] Siek, Jeremy G. ; Vitousek, Michael M. ; Cimini, Matteo ; Boyland, John T.: Refined Criteria for Gradual Typing. In: *1st Summit on Advances in Programming Languages, SNAPL'15, May 3–6, 2015, Asilomar, California, US*, Schloss Dagstuhli, Leibniz-Zentrum für Informatik, 2015 (LIPIcs 32), S. 274–293

[12] Sridharan, Manu ; Chandra, Satish ; Dolby, Julian ; Fink, Stephen J. ; Yahav, Eran: Alias Analysis for Object-Oriented Programs. In: *Aliasing in Object-Oriented Programming.* Springer, 2013 (LNCS 7850), S. 196–232

A Hilfsfunktionen

$$\frac{\texttt{class}\, C\texttt{<}\overline{E}\texttt{>}\, \texttt{extends}\, N\, \{\, \overline{S}\, \overline{g};\, \}\qquad f \notin \overline{g}\qquad \mathit{atype}([\overline{T}/\overline{E}]N, f) = U}{\mathit{atype}(C\texttt{<}\overline{T}\texttt{>}, f) = U}$$

$$\frac{\texttt{class}\, C\texttt{<}\overline{E}\texttt{>}\, \texttt{extends}\, N\, \{\, \overline{S}\, \overline{g};\, \}\qquad f = g_i}{\mathit{atype}(C\texttt{<}\overline{T}\texttt{>}, f) = [\overline{T}/\overline{E}]S_i}$$

$$\frac{\mathit{atype}(N, f) = T}{\mathit{ftype}(N^A, f) = T}\qquad \frac{\mathit{atype}(N, f) = T}{\mathit{ftype}(N^C, f) = T}$$

$$\mathit{dynarg}(\texttt{dynamic})\qquad \frac{\mathit{dynarg}(T_i)}{\mathit{dynarg}(C\texttt{<}\overline{T}\texttt{>})}\qquad \frac{\texttt{class}\, C\texttt{<}\overline{E}\texttt{>}\, \texttt{extends}\, N\, \{\ldots\}\quad \mathit{dynarg}(N)}{\mathit{dynarg}(C\texttt{<}\overline{T}\texttt{>})}$$

$$\frac{\texttt{class}\, C\texttt{<}\overline{E}\texttt{>}\, \texttt{extends}\, N\, \{\, \overline{S}\, \overline{f};\, \}\qquad \mathit{atype}(N, f_i) = U\qquad S_i \neq U}{\mathit{variant}(C\texttt{<}\overline{T}\texttt{>})}$$

$$\frac{\texttt{class}\, C\texttt{<}\overline{E}\texttt{>}\, \texttt{extends}\, N\, \{\ldots\}\qquad \mathit{variant}(N)}{\mathit{variant}(C\texttt{<}\overline{T}\texttt{>})}$$

$$\frac{\mathit{dynarg}(N)}{\mathit{unsafe}(N^A)}\qquad \frac{\mathit{dynarg}(N)}{\mathit{unsafe}(N^C)}\qquad \frac{\mathit{variant}(N)}{\mathit{unsafe}(N^A)}\qquad \frac{\mathit{variant}(N)}{\mathit{unsafe}(N^C)}$$

Abstrakte Interpretation auf domänenspezifischen Sprachen für graphbasierte Optimierungsprobleme

Benjamin Saul, Wolf Zimmermann

Institut für Informatik
Martin-Luther-Universität Halle-Wittenberg
Von-Seckendorff-Platz 1
06120 Halle (Saale)

Zusammenfassung. Domänenspezifische Sprachen erlauben die prägnante Formulierung von Optimierungsproblemen. Ist die Semantik der Sprache klar definiert, so kann diese verwendet werden, um eine abstrakte Interpretation auf den Modellen durchzuführen, die die Problemstellung einschränken oder näher untersuchen können. Dieser Artikel stellt am Beispiel von graphbasierten Optimierungsproblemen vor, wie eine abstrakte Interpretation eingesetzt werden kann.

1 Einleitung

Pumpensysteme transportieren Flüssigkeiten über eine Distanz unter Berücksichtigung von Anforderungen an Druck, Durchfluss und verfügbarem Platz. Durch eine geschickte Auswahl und Kombination verschiedener Pumpen kann bis zu 60% des benötigten Stromverbrauches eingespart werden [8]. Die Aufgabe eines Ingenieurs ist es also, ein möglichst verbrauchsarmes und somit kostenoptimales System zu bestimmen. Abbildung 1 zeigt eine Druckerhöhungsanlage, eine Standardanlage wie sie in Versorgung und Industrie eingesetzt wird.

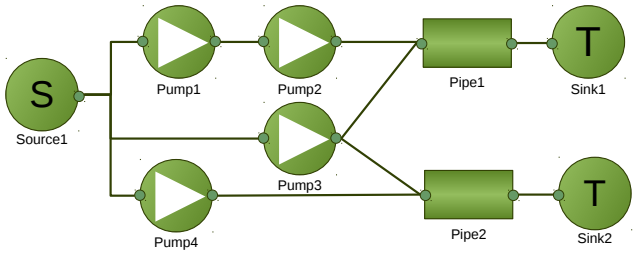


Abb. 1: Beispiel für eine Pumpenanlage, die einen Fluss von einer Quelle zu zwei Senken transportiert.

Die Beispielanlage soll Wasser von einer Quelle zu zwei unterschiedlichen Senken transportieren. Dabei überwinden Rohre die Distanz und Pumpen bauen den Druck auf. Dem Anlagenplaner stehen dabei vier Bauplätze zur Verfügung, in denen Pumpen verschiedenen Typs platziert werden können. Für einen gegebenen Ausgangsdruck in der Quelle und geforderten Drücken und Volumenströmen in den Senken müssen nun passende Pumpen ausgewählt werden. Andere Armaturen werden zur Vereinfachung vernachlässigt.

Das Auswahlverfahren der bestmöglichen Pumpen kann als gemischt-ganzzahliges lineares Programm (MILP) formuliert werden [9]. Dessen Lösung entspricht dem verbrauchsärmsten Pumpensystem, welches die gegebenen Randbedingungen erfüllt. Um diese Modellierungsmethoden besser anwendbar zu machen, wurde die domänenspezifische Sprache SHEP – Sprache für hocheffiziente Pumpensysteme – entwickelt [10]. Diese erlaubt die Beschreibung der Randbedingungen eines Pumpensystemes in einer knappen und präzisen Form. Die Lösung des MILPs geschieht durch bestehende Löser für diese Probleme, das Optimierungsergebnis wird anschließend noch ausgewertet. Abbildung 2 zeigt den Ablauf der Werkzeugkette.

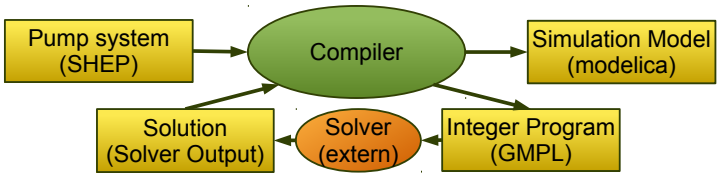


Abb. 2: Werkzeugkette für die Berechnung verbrauchsoptimaler Pumpensysteme.

Die Nutzung externer Löser hat den Vorteil, dass diese leicht austauschbar sind und hier keine Neuentwicklungen bestehender Algorithmen geschehen müssen. Allerdings haben solche Löser zwei Nachteile. Zum einen kann die Lösungszeit sehr lang sein, zum anderen werden inkonsistente Systeme nur schwer nachvollziehbar detektiert.

Eine domänenspezifische Sprache beinhaltet Informationen über Zusammenhänge, welche sonst im linearen Programm erst wieder interpretiert werden müssen. So ist zum Beispiel der Systemgraph und die verschiedenen Abhängigkeiten der Komponenten bekannt. Mit diesen Informatiionen kann eine Programmanalyse in Form einer abstrakten Interpretation auf dem SHEP Modell durchgeführt werden. Dabei sollen die folgenden Fragen beantwortet werden:

- Wie funktioniert eine abstrakte Interpretation auf domänenspezifischen Sprachen für graphbasierte Optimierung?
- Bringen die Ergebnisse der Analyse einen Vorteil in Bezug auf Problemgröße und Anwenderfreundlichkeit?

Um diese Fragen zu untersuchen, wird zunächst in Kapitel 2 die Pumpendomäne vorgestellt. In Kapitel 3 wird anschließend die domänenspezifische Sprache SHEP und ihre Semantik vorgestellt. Darauf aufbauend wird in Kapitel 4 ein Framework für eine abstrakte Interpretation erarbeitet und untersucht. Kapitel 5 stellt einige verwandte Arbeiten vor. Kapitel 6 fässt die Ergebnisse dieser Arbeit zusammen und schließt mit einem Ausblick ab.

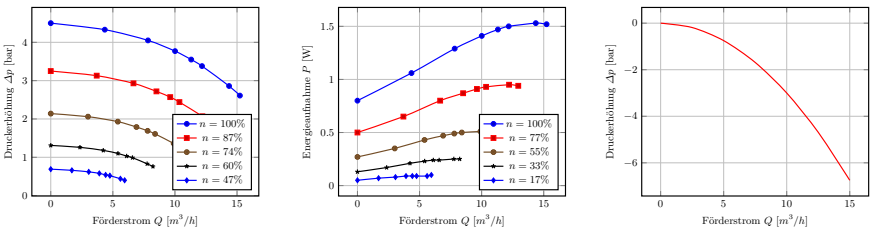
2 Domänenspezifisches Verhalten

Komponenten in Pumpensystemen haben ein unterschiedliches Verhalten. Allerdings lassen sich Gruppen von Komponenten mit ähnlichem Verhalten identifizieren. Leitende Komponenten, wie Pumpen und Rohre, leiten den Volumenstrom von ihrem *in*-Anschluss zum *out*-Anschluss. Dabei wird Volumen vom Förderstrom erhalten, dessen Richtung mit einem Vorzeichenwechsel angepasst und der Druck erhöht bzw. reduziert. Die Größe der Druckdifferenz ist dabei i. A. abhängig vom Volumenstrom, woraus sich

$$Druck_{out} = Druck_{in} + \Delta p(Fluss_{in}) \qquad \text{und} \tag{1}$$

$$Fluss_{in} = -Fluss_{out} \tag{2}$$

ergeben. Im Falle von Pumpen hängt Δp auch von deren Drehzahl ab. Durch eine Erhöhung der Rotirendrehzahl erhöht sich auch die Druckerhöhung auf Kosten des Energieverbrauches der Pumpe. Der genaue Zusammenhang zwischen den verschiedenen Größen wird durch Kennlinien spezifiziert. Kennlinien sind gemessene und interpolierte Funktionen und werden als Funktionsgraphen vom Hersteller veröffentlicht. Sie beschreiben, wie viel Druck eine Pumpe aufbauen kann, wenn sie mit einer bestimmten Drehzahl eingestellt ist und ein gewisser Förderstrom anliegt. Abbildung 3a zeigt aus [6] entnommene Kennlinien.



(a) Die Kennlinie einer Pumpe stellt den Zusammenhang zwischen Druckaufbau Δp, Förderstrom Q, Drehzahl n und Energieaufnahme P dar.

(b) Druckverlust durch ein Rohr, dargestellt als Kennlinie.

Abb. 3: Verhaltensfunktionen von Pumpen und Rohren.

Während bei Pumpen eine einzelne Pumpe durch die veränderliche Drehzahl verschiedene Kennlinien aufweist, besitzen Rohre nur eine einzelne Kennlinie.

Diese ist Abhänig vom Durchmesser, der Länge, der Lage und verschiedener Materialkonstanten und wird durch die Formel

$$\Delta p(Fluss) = -c \cdot \frac{Länge \cdot Fluss^2}{Durchmesser^3} - Höhendifferenz \tag{3}$$

beschrieben [7]. Dabei fässt die Konstante c mehrere Faktoren zusammen, auf die hier nicht näher eingegangen werden soll. Abbildung 3b zeigt den Druckabfall eines Rohrs als Kennlinie.

Wenn Komponenten über ihre Anschlüsse miteinander verbunden werden, so wird der darüber transportierte Volumenstrom auf die verschiedenen Anschlüsse – unter Berücksichtigung der Fließrichtung – aufsummiert. Drücke an verbundenen Stellen müssen gleich sein, da jeder Stelle im Pumpensystem genau ein Druck zugeordnet wird. Diese Zusammenhänge werden durch

$$\forall_{p \in Anschlüsse} : Fluss_p = \sum_{(a,p) \in Kanten} Fluss_{ap} - \sum_{(p,a) \in Kanten} Fluss_{pa} \quad \text{und} \tag{4}$$

$$\forall_{(a,b) \in Kanten} : Druck_a = Druck_b \tag{5}$$

beschrieben. Dabei wird zwischen dem durch einen Anschluss fließendem und dem über eine Verbindung geleiteten Volumenstrom unterschieden. Eine Kante im System entspricht hier der Verbindung zweier Komponenten, wobei eventuelle weitere Armaturen, welche im gebauten System notwendig sind, der Einfachheit halber nicht modelliert werden.

3 Syntax und Semantik von SHEP

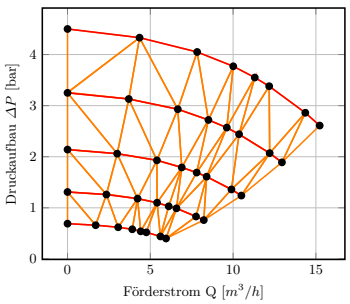
Die domänenspezifische Sprache SHEP beschreibt Pumpensysteme, ihre Komponenten und verschiedene Lastprofile, um damit eine Systemoptimierung durchzuführen [10]. Dazu können Komponententypen definiert werden. Die Definition umfasst vor allem die Beschreibung der Kennlinie, die das Verhalten der Komponente definiert. In Codeausschnitt 1 wird die in Abb. 3a gezeigte Kennlinie modelliert.

```
pump Movitec scales to {34,47,60,74,87,100}
characteristic
  speed flow  head  power;
  100   0.00  4.50  0.80;
  100   4.35  4.33  1.06;
  100   7.83  4.05  1.29;
  // ...
  ports
    in  Flange;
    out Flange;
end
```

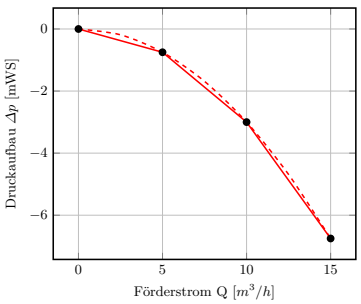
Codeausschnitt 1: SHEP Modell der Movitec Pumpe aus Abb. 3a.

Eine Pumpe besitzt einen Typnamen, eine Kennlinie und zwei Anschlüsse. Basierend auf einer Kennlinie bei einer bestimmten Drehzahl, i. d. R. die maximale Drehzahl (100%), können die Kennlinien anderer Drehzahlen durch Ähnlichkeitsgesetze berechnet werden indem die gegebenen Datenpunkte skaliert werden [7]. Die Definition der Anschlüsse ist notwendig, damit beim Verbinden bestimmt werden kann, ob diese zueinander passen.

Die Berechnung der Druckänderung Δp erfolgt durch Interpolation der vorhandenen Datenpunkte. Mehrere Methoden sind dafür möglich, wobei sich innerhalb dieser Arbeit auf eine Triangulation geeinigt wurde, die bereits in [9] verwendet wurde. Dabei wird eine Triangulierung über alle Datenpunkte nach festen Regeln bestimmt, sodass jeweils drei Datenpunkte ihren Zwischenraum interpolieren. Für diese Modellierung sind mehrere ganzzahlige Entscheidungsvariablen notwendig, welche sich negativ auf die spätere Lösungszeit auswirken. Abbildung 4a zeigt die Triangulierung der Kennlinie aus Abb. 3a.



(a) Triangulierung der Kennlinie aus 3a.



(b) Linearisierung (durchgängig) des Druckabfalls in Rohren (gestrichelt, vgl. Abb. 3b.

Abb. 4: Linearisierung von Kennlinien für Pumpen und Rohre.

Rohre und ihr Verhalten werden durch ihre Länge und den Durchmesser bestimmt. Daher sind diese in der Spezifikation anstelle einer expliziten Kennlinie zu finden (Codeabschnitt 2).

```
pipe CommonPipe with 20 support points
  flow     = [0,100];
  diameter = {20, 40, 60} [cm];
  length   = {[20,35], [40,60], [80,100]} [m];
  ports
    in  Flange;
    out Flange;
end
```

Codeausschnitt 2: Declaration of a Pipe in SHEP

In der Typdefinition werden nur mögliche Längen festgelegt, die tatsächliche Länge wird bei jedem Rohr einzeln festgelegt und muss sich an die Intervalle der Typen halten. So kann ein Rohr vom Typ *CommonPipe* nur Längen zwischen 20 und 35, 40 und 60 oder 80 und 100 Metern annehmen.

Um den nichtlinearen Druckabfall (3) zu linearisieren, sind Sützsstellen notwendig. Diese werden in gegebener Anzahl über den zuvor bestimmten Wertebereich verteilt. Sollten keine Angaben diesbezüglich im Modell enthalten sein, so werden die Stützstellen vom Übersetzer nach gewissen Kriterien wie Genauigkeit bestimmt. Ein Beispiel für eine Interpolation des Druckabfalls in Rohren ist in Abb. 4b gegeben.

Durch die Linearisierung wird der mögliche Druckabfall überschätzt. Dadurch wird der lineare Lösungsraum gegenüber dem Nichtlinearen verkleinert, wodurch Lösungen ausgeschlossen werden könnten. Allerdings wird so verhindert, dass es Lösungen im linearen Lösungsraum gibt, welche im nichtlinearen Raum nicht verwendet werden können.

Das Pumpensystem selbst besteht aus den zu kombinierenden Komponenten, den Verbindungen zwischen diesen und diversen Lastfällen, wie Tag- und Nachtbetrieb. Codeabschnitt 5 zeigt die Spezifikation des Pumpensystems aus Abb. 1.

```
system MySys
 Source      Source1;
 Movitec     Pump1 optional;
 Movitec     Pump2 optional;
 Movitec     Pump3 optional;
 Movitec     Pump4 optional;
 CommonPipe Pipe1 (length = 3 [m]);
 CommonPipe Pipe2 (length = 5 [m]);
 Sink        Sink1, Sink2;
connections
 Source1 → Pump1 → Pump2;
 Pump2   → Pipe1 → Sink1;
 Source1 → Pump3 → Pipe1;
 Pump3   → Pipe2 → Source2;
 Source1 → Pump4 → Pipe2;
 Pipe2   → Sink2;
objective
 minimize costs
  (powerprice=0.24);
scenarios
scenario daytime
  weight 20000 [hours]:
 Source1.out.press = 0 [bar]  ;
 Sink1.in.flow     = 3 [m^3/h];
 Sink1.in.press    = 4 [bar]  ;
 Sink2.in.flow     = 5 [m^3/h];
 Sink2.in.press    = 3 [bar]  ;
end
```

Abb. 5: Beschreibung des Pumpensystems aus Abb. 1 in SHEP.

Aus dieser Spezifikation schließlich wird das lineare Programm bzw. das Gleichungssystem gebildet. Für jede angelegte Komponente werden die ihrem Typ entsprechenden Gleichungen generiert. Dies verbindet ihren eingehenden mit dem ausgehenden Anschluss.

Durch die Verbindungen – beschrieben durch den $\rightarrow$-Operator – wird dann der Systemgraph aufgebaut. Die Verteilungsgleichungen (4) und (5) werden basierend auf den hier beschriebenen Kanten implementiert.

Eine Zielfunktion *objective* gibt an, wonach optimiert werden soll. Im Allgemeinen werden die Gesamtkosten, also Einkaufskosten und Stromverbrauch, minimiert, oder nur der Stromverbrauch über die betrachteten Lastfälle.

Ein Lastfall, oder auch *scenario*, wird durch eine Menge an Forderungen an Komponenten wie Quellen und Senken sowie einer erwarteten Betriebszeit definiert. Die Forderungen werden als Gleichung formuliert und können so direkt in das lineare Programm übernommen werden.

4 Abstrakte Interpretation

Aufgabe eines Pumpensystems ist die Erfüllung von Anforderungen an Drücken und Volumenströmen an verschiedenen Stellen im System. Erreichbare Werte definieren dabei das sogenannte Betriebsfeld einer Anlage bzw. einer Pumpe.

Definition 1. *Das* ***Betriebsfeld*** $\Omega \subseteq \mathbb{R}^2$ *eines Knotenpunktes in einem Pumpensystem gibt an, welche Drücke und Durchflüsse dort erreicht werden können. Für eine Pumpe gibt das Betriebsfeld den Bereich ihrer Kennlinien an.*

Alleinige Betrachtung der Betriebsfelder, also Vernachlässigung von Parametern wie Drehzahl oder Stromverbrauch, genügt für die Erfüllbarkeitsfrage von Pumpenanlagen. Die dadurch entstehende Abstraktion ist in Abb. 6 dargestellt.

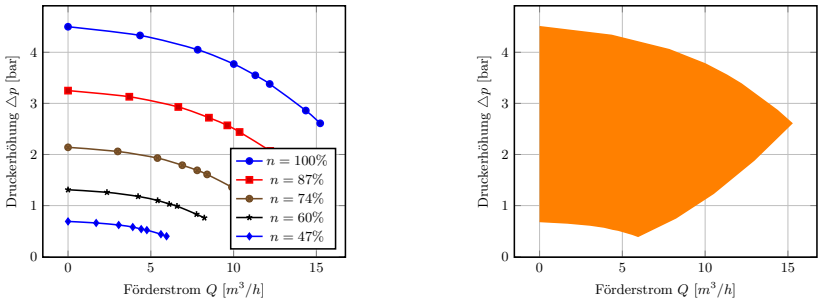
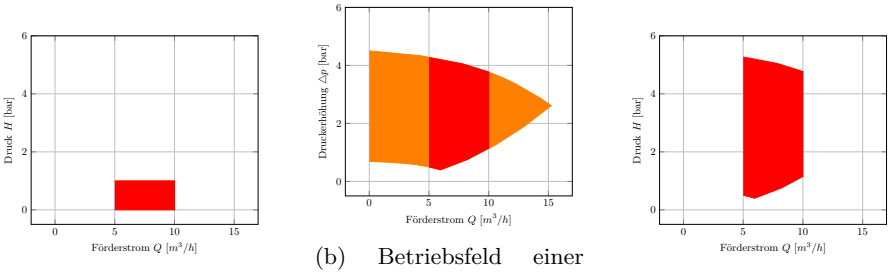


Abb. 6: Abstraktion auf das Betriebsfeld einer Pumpe durch Vernachlässigung der Kennlinien.

Die Druckänderung lässt sich über die auftretenden Mengen definieren. Für ein Pumpenbetriebsfeld $\triangle \in \Omega$ und ein eingehendes Betriebsfeld $X \in \Omega$ definieren wir die Druckänderung $P_\triangle$ entsprechend (1) und (2) als

$$P_\triangle(X) := \{(x_1, x_2 + p) : (x_1, x_2) \in X, (x_1, p) \in \triangle\}.$$

Hierbei ist zu beachten, dass die entstehende Menge auch leer sein kann. Pumpen dürfen nicht außerhalb ihrer Kennlinien arbeiten. In Abbildung 7 wird dieser Zusammenhang veranschaulicht.



(a) Eingehendes Betriebsfeld für eine Pumpe.

(b) Betriebsfeld einer Pumpe mit hervorgehobenen Werten aus Abb. 7a.

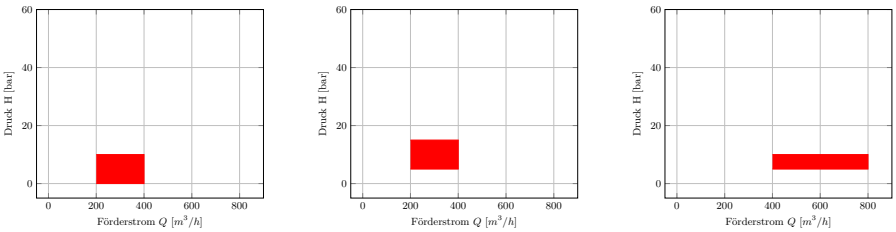
(c) Ausgehendes Betriebsfeld an der Pumpe.

Abb. 7: Druckerhöhung einer Pumpe am Beispiel.

Während Komponenten Druck verändern und den Förderstrom erhalten, ist dies bei den Verbindungen zwischen Komponenten genau anders herum. Wie in (5) und (4) beschrieben, teilen sich die Volumenströme auf während der Druck gleichgesetzt wird. Der Operator $\uplus$ beschreibt diesen Zusammenhang. Für zwei Betriebsfelder $X, Y \in \Omega$ wird definiert:

$$X \uplus Y := \{(p, x_2 + y_2) : p \in \mathbb{R}, (p, x_2) \in X, (p, y_2) \in Y\}.$$

Dabei gilt ähnlich zur Druckerhöhung, dass nur bei gleichem Druck sich die Volumenströme addieren können. Abbildung 8 verdeutlicht diesen Operator.



(a) A (b) B (c) $A \uplus B$.

Abb. 8: Zusammenführen zweier Betriebsfelder an einem Punkt.

Mit den oben definierten Funktionen und Operatoren ist es möglich, aus den entsprechenden Codezeilen in SHEP ein Gleichungssystem zu konstruieren. Dies wird in Abb. 9 gezeigt.

```
Movitec Pump1,Pump2,
        Pump3,Pump4;
CPipe Pipe1, Pipe2;
Source1 → Pump1;
Pump1   → Pump2;
Pump2   → Pipe1;
Pipe1   → Sink1;
Source1 → Pump3;
Pump3   → Pipe1;
Pump3   → Pipe2;
Pipe2   → Source2;
Source1 → Pump4;
Pump4   → Pipe2;
```

(a) Komponentendefinitionen und Kanten aus dem Codeabschnitt 5.

$$
\begin{aligned}
Source1.out &= \text{init} \\
Pump1.out &= P_{Movitec}(Pump1.in) \\
Pipe1.out &= P_{CPipe}(Pipe1.in) \\
Pump1.in &= S.out \\
Pump2.in &= Pump1.out \\
Pipe1.in &= Pump2.out \uplus Pump3.out \\
Pump3.in &= S.out \\
Pipe2.in &= Pump3.out \uplus Pump4.out \\
Pump4.in &= S.out \\
Sink1.in &= Pipe1.out \\
Sink2.in &= Pipe2.out
\end{aligned}
$$

(b) Gleichungssystem

Abb. 9: Gegenüberstellung von SHEP Modell und Gleichungssystem.

Die Lösung dieses Gleichungsystems ergibt sich durch Iteration. Als Startwert wird dabei Ω verwendet. Die Terminierung des Verfahrens lässt sich durch Anwendung des Fixpunktsatzes von Kleene zeigen.

Lemma 1 (Kleenes Fixpunktsatz [5]). *Sei $(L, \sqsubseteq)$ ein vollständige Halbordnung mit einem kleinstem Element, und sei $f : L \to L$ eine monotone Funktion. Dann hat f einen kleinsten Fixpunkt, welcher als das Supremum der aufsteigenden Kleene Kette von f ist.*

Die Berechnung über den $\uplus$-Operator führt zu einer Überschätzung der tatsächlich möglichen Betriebsfelder, da immer nur direkt verbundene Anschlüsse betrachtet werden. Dadurch wird nicht bestimmt, ob ein Pumpensystem so tatsächlich eine alle Anforderungen erfüllende Konfiguration besitzt. Sollte sich bei der Fixpunktiteration die leere Menge in einer Senke ergeben, so heißt dass, dass es keine gültige Lösung für das Pumpensystem geben kann. Die erforderlichen Drücke bzw. Durchflüsse können an bestimmten Stellen nicht erreicht werden. Diese Information kann an den Nutzer weitergegeben werden, um fehlerhafte Modellierung zu markieren. Wird ein leeres Betriebsfeld an einer anderen Komponenten berechnet, so kann diese Komponente ohne Verlust aus dem System entfernt werden, da sie so nicht betrieben werden kann.

Theorem 1. *Das durch die Funktion $P_\triangle$ und den Operator $\uplus$ Gleichungssystem definiert auf der Menge der Betriebsfelder eine abstrakte Interpretation.*

Beweis. Für den Beweis wird gezeigt, dass $(\Omega, \subseteq)$ ein vollständiger Verband ist und $P_\triangle$ sowie $\uplus$ monotone Funktionen bzw. Operatoren darauf sind. Durch Anwendung von Lemma 1 folgt daraus, dass das Gleichungssystem genau eine Lösung hat, welche nach Konstruktion die möglichen Betriebsfelder berechnet.

Die Grundmenge $\Omega = \mathbb{R}^2$ bildet unter $\subseteq$ einen vollständigen Verband mit Supremum $\mathbb{R}^2$ und Infimum $\emptyset$, da Reflexivität, Transitivität und Asymmetrie durch die Teilmengenbeziehung bereits gegeben sind.

Die Funktion $P_\triangle$ ist monotone: für alle $x, y \in \Omega$ und $x \subseteq y$ gilt

$$\begin{aligned} P_\triangle(x) &= \{(x_1, x_2 + p) : (x_1, x_2) \in X, \qquad (x_1, p) \in \triangle\} \\ &\subseteq \{(x_1, x_2 + p) : (x_1, x_2) \in X \cup Y, (x_1, p) \in \triangle\} \\ &= P_\triangle(y). \end{aligned}$$

Außerdem gilt für den Operator $\uplus$ und alle $x^1, x^2, y^1, y^2 \in \Omega$ mit $x^1 \subseteq x^2$ und $y^1 \subseteq y^2$

$$\begin{aligned} x^1 \uplus y^1 &= \{(p, x + y) : p \in \mathbb{R}, (p, x) \in x^1, (p, y) \in y^1\} \\ &\subseteq \{(p, x + y) : p \in \mathbb{R}, (p, x) \in x^2, (p, y) \in y^2\} \\ &= x^2 \uplus y^2. \end{aligned}$$

Somit sind die Anforderungen für Lemma 1 erfüllt und das Gleichungssystem konvergiert gegen einen Fixpunkt. □

5 Verwandte Arbeiten

Abstrakte Interpretationen wurden in [3] vorgestellt. Darin wird das Framework für eine allgemeine abstrakte Interpretation auf Programmiersprachen vorgestellt, welches vielseitige Anwendung in Programmanalysen findet ([2], [1]).

In [2] wird die abstrakte Interpretation genutzt um das Programmverhalten bzgl. Aussagen in temporaler Logik zu überprüfen. Durch Abstraktion der zu beobachtenden Aussagen können auch größere Eingaben getestet werden.

Alur et. al. beschreiben in [1] eine abstrakte Interpretation für die Untersuchung hybrider Systeme. Dabei müssen kontinuierliche Domänen diskretisiert werden, um sie zu analysieren. Dabei müssen einige Wertebereiche zusätzlich eingeschränkt werden, um eine Entscheidbarkeit zu erzwingen.

Götz et. al. zeigen einen Ansatz zur optimalen Verteilung von Lastprofilen, bei dem die eigentliche Optimierung über ein generiertes gemsicht-ganzzahliges lineares Programm erfolgt [4]. Auch hier wird versucht, die Lösungszeit durch entsprechende Analysen zu verringern.

Für die effiziente Lösung von gemischt-ganzzahligen linearen Programmen gibt es verschiedene Ansätze, die von der Problemstruktur und der Größe abhängen [11]. Diese Methoden werden in dieser Arbeit nicht weiter verfolgt, da sie unabhängig von der abstrakten Interpretation auf den generierten linearen Programmen erfolgen können.

6 Zusammenfassung und Ausblick

Diese Arbeit zeigt ein Framework für eine abstrakte Interpretation auf einer domänenspezifischen Sprache für die Optimierung von Pumpensystemen. Ein Pumpensystem ist dabei als Graph zu verstehen, bei dem die Komponenten Funktionen zwischen ihren beiden Anschlüssen definieren und die durch Kanten miteinander über gewisse Variablen verbunden sind. Die wichtigsten Variablen dabei sind Druck und Durchfluss, die auch für die Erfüllbarkeit eines Pumpensystems entscheiden. Nur wenn erforderliche Größen erreicht werden, kann das System so funktionieren.

Die eigentliche Optimierung dabei geschieht über einen externen Löser für gemischt-ganzzahlige lineare Programme, welche aus der dafür entwickelten Sprache SHEP generiert werden. Die Nachteile eines Lösers sind dabei die eventuell lange Rechenzeit sowie die schlechte Markierung von Inkonsistenzen. Diese Schwächen versucht die hier entwickelte Analyse zu beheben.

Die Abstrakte Interpretation analysiert mögliche Betriebsfelder und bestimmt so für jeden Knotenpunkt im Pumpengraphen die erreichbaren Werte von Drücken und Durchflüssen. Diese Information kann einerseits verwendet werden, um Inkonsistenzen zu erkennen. Ist ein geforderter Druck zum Beispiel nicht innerhalb des jeweiligen Betriebsfeldes, so kann die Anlage die gestellten Anforderungen nicht erfüllen. Andererseits müssen die Funktionen der einzelnen Komponenten nur für die überhaupt erreichbaren Wertebereiche modelliert werden. Dadurch können Variablen eingespart werden.

Die vorgestellten Analysen sind noch erweiterbar. Zusätzlich zu den vorwärts gerichteten Berechnungen kann für jede Pumpe beispielsweise auch die umgekehrte Funktion bestimmt werden, um den Zusammenhang zwischen Eingang und Ausgang herzustellen. Damit lassen sich deren Wertebereiche weiter einschränken. Außerdem wurden optionale Komponenten, also solche, die durch die Optimierung entfallen können, nicht weiter betrachtet.

Danksagung

Dieses Projekt wird gefördert durch das Bundesministerium für Wirtschaft und Technologie unter dem Förderkennzeichen 03ET1134C. Partner in diesem Projekt sind die Technische Universität Darmstadt und die KSB Aktiengesellschaft.

Literatur

[1] Rajeev Alur u. a. „Discrete abstractions of hybrid systems“. In: *Proceedings of the IEEE* 88.7 (2000), S. 971–984.

[2] Edmund M Clarke, Orna Grumberg und David E Long. „Model checking and abstraction“. In: *ACM transactions on Programming Languages and Systems (TOPLAS)* 16.5 (1994), S. 1512–1542.

[3] Patrick Cousot und Radhia Cousot. „Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints“. In: *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM. 1977, S. 238–252.

[4] S Götz u. a. „Modeldriven self-optimization using integer linear programming and pseudoboolean optimization“. In: *Proceedings of ADAPTIVE* (2013), S. 55–64.

[5] Stephen Cole Kleene u. a. *Introduction to metamathematics*. Bd. 483. van Nostrand New York, 1952.

[6] KSB Aktiengesellschaft. *Baureihenheft Movitec B*. 2013. URL: http://www.pumpentechnik-strauss.de/index_htm_files/Movitec50.pdf.

[7] AG KSB. „Kreiselpumpen Lexikon“. In: *KSB AG, Frankenthal, Germany* (1989).

[8] Manfred Oesterle und Fred Leidig. „Methodisch sichere, schnelle Produktionsanlaufe in der Mechatronik (MESSPRO)“. In: *Schneller Produktionsanlauf in der Wertschopfungskette* Band 2 (2007).

[9] Peter Pelz u. a. *Designing Pump Systems by Discrete Mathematical Topology Optimization: The Artificial Fluid Systems Designer*. Techn. Ber. Düsseldorf: International Rotating Equipment Conference, 2012.

[10] Benjamin Saul, Christian Berg und Wolf Zimmermann. „A Domain Specific Language for Optimal Pumping Systems“. In: *Proceedings of the 1st Industry Track on Software Language Engineering*. ITSLE 2016. Amsterdam, Netherlands: ACM, 2016, S. 23–32. ISBN: 978-1-4503-4646-7. DOI: 10.1145/2998407.2998412. URL: http://doi.acm.org/10.1145/2998407.2998412.

[11] Martin WP Savelsbergh. „Preprocessing and probing techniques for mixed integer programming problems“. In: *ORSA Journal on Computing* 6.4 (1994), S. 445–454.

JavaTX
– Extended abstract –

Andreas Stadelmeier
Martin Plümicke

Baden-Wuerttemberg Cooperative State University Stuttgart/Horb
Department of Computer Science
Florianstraße 15, D–72160 Horb
stan@hb.dhbwstuttgart.de
pl@dhbw.de

Abstract. JavaTX steht für Java Type Extended und ist eine Verbindung aus Java und Typinferenz.
Unser Projekt zielt darauf ab die objektorientierte Programmiersprache Java mit einem globalen Typinferenzalgorithmus auszustatten. Dieser erlaubt dem Benutzer sämtliche Typen in seinem Programmcode auszulassen ohne dabei auf die Vorteile einer statisch typisierten objektorientierten Programmiersprache verzichten zu müssen.
Zuletzt führte das JavaTX-Projekt einen Code-Rewrite durch, bei dem der komplette Programmcode von Grund auf neu geschrieben wurde.
Der Vortrag präsentiert das Ergebnis dieses Rewrites und die damit einhergehenden Änderungen und neuen Funktionalitäten des JavaTX-Compilers.

1 JavaTX

Das JavaTX Projekt entwickelt Compiler und Werkzeuge für die Programmiersprache JavaTX. Bei JavaTX handelt es sich um eine Abwandlung der Programmiersprache Java, welche es erlaubt Typen im Quellcode auszulassen.
Das Projekt hat das Ziel die Sprache Java bestmöglich zu imitieren und die meisten seiner Sprachfeatures zu übernehmen. Zusätzlich soll dieser Compiler mit einem Typinferenzalgorithmus gekoppelt werden, welcher im Quellcode ausgelassene Typen selbstständig ermittelt. Dadurch soll es möglich sein Java ähnlich einer dynamischen Programmiersprache ohne die Angaben von Typen zu benutzen, aber immer noch die Vorteile einer statisch typisierten Programmiersprache zu genießen.
Diese Ziele versucht das Projekt mittels eines eigenständigen Compilers zu realisieren. Dieser ist in Java geschrieben und wird weitestgehend über Studienarbeiten vorangetrieben.

1.1 Typinferenz

Es handelt sich bei der von uns verwendeten Typinferenz um globale Typinferenz. Das bedeutet, dass der Algorithmus auch dann zurecht kommt, wenn der Programmierer im gesamten Projekt keinen einzigen Typ angegeben hat.

Bisher wird in der Praxis für Objektorientierte Programmiersprachen lokale Typinferenz eingesetzt. Im Falle von Java ist dies zum Beispiel beim Einsatz des Diamant-Operators wie zum Beispiel in
`List<String> listOfStrings = new ArrayList<>();`
der Fall.
Die Abbildungen 1 und 2 zeigen den Einsatz unseres Typinferenzalgorithmus. Alle in Abbildung 1 grau gefärbten Angaben würden von unserem JavaTX Compiler intern automatisch hinzugefügt und müssten vom Programmierer nicht angegeben werden.

```
interface Fun1{
    A apply(B par1);
}

interface Fun2<A,B,C>{
    A apply(B par1, C par2);
}

class Matrix<A, B>{
    Fun1<Fun1<A, Fun2<A, B, Matrix>>,B>
    op = (m) → (f) → f.apply(m, this);
}
```

Fig. 1. Java **ohne** Typinferenz

```
class Matrix{
    op = (m) -> (f) -> f.apply(m, this);
}
```

Fig. 2. Java **mit** Typinferenz

1.2 Code Rewrite

Die Studienarbeiten, welche den Compiler weiterentwickelten, beschränken sich auf einen Arbeitsaufwand von gerade einmal 150 Stunden. Das führt dazu, dass immer in kleinen Teilprojekten am Compiler gearbeitet wurde, was zu einem chaotischen Zustand des Projekts geführt hat. Ein kompletter Rewrite des Quellcodes war notwendig.
Da der Compiler von Grund auf neu geschrieben nutzen wir dies zum Umbau grundlegender Strukturen. Die dabei entstandenden Änderungen, Verbesserungen und neuen Funktionen sind im folgenden aufgelistet.

Parser mit voller Java 8 Unterstützung: Das Projek stieg auf ANTLR als Parser Generator um. Dieser bringt eine vollständige Grammatik für Java 8 mit sich. Diese musste für JavaTX nur leicht angepasst werden. Der neue Parser unterstützt daher die Sprache Java komplett und akzeptiert zusätzlich dazu auch Java Programme, in denen Typen ausgelassen wurden. Der restliche Teil des JavaTX Compilers unterstützt zwar noch nicht alle Sprachfeatures von Java, hat nun mit diesem Parser allerdings eine starke Basis auf die zukünftige Erweiterungen aufbauen können.

Auftrennung des Compiliervorgangs in einzelne Schritte: *Divide & Conquer* ist eines der entscheidenen Programmierparadigmen vor allem bei Objektorientierten Programmiersprachen wie Java. In unserem Java Projekt sind allerdings über die Zeit einige Aufgaben und Algorithmen zusammengewachsen. Das führte zu einer wirren Struktur und nicht mehr nachvollziehbarem Ablauf des Compiliervorgangs. Der Code Rewrite gab die Chance für eine Umstrukturierung. Das Projekt wurde in verschiedene Aufgaben unterteilt, die alle aufeinander aufbauen und sequentiell abgelaufen werden. Dabei achteten wir auf eine möglichst strikte Trennung der Aufgaben. Das erleichtert das Arbeiten an den einzelnen Aufgaben, da nur dieser eine Teil im Detail verstanden werden muss und nicht das ganze Projekt. Außerdem lassen sich nun Teile des Algorithmus leichter austauschen. Zudem versuchten wir mit der Einführung gewisser Vorgaben, wie dem Immutable AST und der Verwendung des Listener-Patterns, diese Struktur auch für zukünftige Projekte einzuhalten.

Transformations of CSPs to Regular CSPs

Sven Loeffler, Ke Liu, Petra Hofstedt

Brandenburg University of Technology Cottbus-Senftenberg, Germany

Abstract. With the development of constraint programming (CP), NP-complete problems can be molded and solved in a powerful declarative way. Until today, the most and with biggest success researched problems in constraint programming are rostering, graph coloring and satisfiability (SAT) problems [4].

In an ideal declarative programming world, the program execution speed of a constraint problem does not depend on the syntax of semantically equivalent problem descriptions. However, as far as we know, this is not the reality for general constraint problems.

Generally, there are several possible manners to model a given constraint problem since the problem can be described from different perspectives, therefore, two or more different sets of constraints, which might consist of various constraints, could be used to attack the same problem. The resulting constraint problems can lead to very different execution speed and execution behavior. The main reason for this is that there are different constraints with several propagation algorithms, which can describe the same restrictions. Consequently, the replacement of constraints by constraints of another type (in this case regular constraints) and the combination of multiple constraints to a new constraint can lead to a faster solution finding process and a removement of undesirable redundancy.

In our context, undesirable redundancy means the redundancy which occurs through the use of similar or overlapping constraints which decrease the solution speed. In contrast to this, redundancy is often used in a targeted manner to improve the solving speed of a constraint problem. We call this positive redundancy. The mix of removing undesirable redundancy and adding positive redundancy might also be a topic of future research, but until yet we consider only the removement of undesirable redundancy.

Constraint Satisfaction Problems (CSP). The goal of our work is to improve the solving speed of a Constraint Satisfaction Problem (CSP) which is defined in the following way.

Definition 1. ***CSP*** *[1] A constraint satisfaction problem (CSP) is defined as a 3-tuple* $P = (X, D, C)$ *with*

$X = \{x_1, x_2, \ldots, x_n\}$ *is a set of variables,*
$D = \{D_1, D_2, \ldots, D_n\}$ *is a set of finite domains where* D_i *is the domain of* x_i*,*
$C = \{c_1, c_2, \ldots, c_m\}$ *is a set of primitive or global constraints containing between one and all variables in* X*.*

We transform several kinds of (global and sets of local) constraints into regular membership constraints and replace then the original constraints by the created regular constraints in a CSP. For selected problems (like rostering problems) we increased the solution speed of CSPs, removed undesirable redundancy and decreased the number of unnecessary backtracks by the replacement of some or all constraints of a CSP with regular constraints followed by the combination of such created multiple regular constraints into a new, more precise regular constraint. The regular constraint and its propagation algorithm [6,5,2] provide on the basis of this approach.

Regular transformations. According to our definition, a CSP P has a fixed number n of variables $X = \{x_1, \ldots, x_n\}$ and the domains $D = \{D_1, \ldots, D_n\}$ of these variables are finite, it follows that the potential solution space of the CSP P is limited by l (see Eq.1).

$$l = \prod_{i=1}^{n} |D_i|. \tag{1}$$

A limited number of solutions can be enumerated by a regular language, and based on the Myhill-Nerode theorem [3] every regular language has at least one automaton which describes this language. A consequence of this is that for every CSP P at least one regular CSP P_{reg} exists which is declaratively equivalent to P and contains only one regular constraint. We name such a CSP, which only contains (one or more) regular constraints, a regular CSP.

If all solutions of a CSP P are known, then such a regular CSP $P_{reg} = (X, D, C_{reg})$ can generated by the enumeration of all solutions of P. In practice, however, this is not useful because we intend to use the regular CSP P_{reg} to find one or all solutions of P faster.

This is the reason, why we are interested in direct transformations of constraints into regular constraints. For some global constraints, among them *count*, *global cardinality constraint*, and *table* constraints we defined corresponding efficient transformations. In addition to this we defined a transformation for sub-CSPs (in particular binary ones) over sub-sets of variables $X' \subseteq X$ and constraints $C' \subseteq C$ of a CSP $P = (X, D, C)$.

After the transformation into regular constraints it is possible to use the automaton methods *intersection* and *minimize* to combine different automatons to a new automaton and also different constraints to one new constraint. By doing this, redundancy between to constraints can be removed. This approach allows it, especially, to combine constraints of original different kinds together, for example *count* with *table*. In our test cases we have experienced a significant reduction of the size of the necessary search tree and a great improvement of the solving speed of CSPs.

Summary. We have presented a new approach for the optimization of general CSPs using the regular constraint.

It has shown that a general CSP can be transformed into a regular CSP and some special global constraints can be transformed directly efficiently into regular constraints. By the use of automata intersection and minimization regular constraints can be minimized. The evaluation of this approach by a rostering example has shown a significantly reduction of the size of the search tree and a great improvement of execution time.

Our investigations support that our approach can be applied successfully when considering sub-problems of a potentially large CSP P and, thus, subsets of its variables X. The transformation of only sub-problems (instead of P completely) is, thus, much faster and, nevertheless, leads to a reduction of the number of constraints (also the number of backtracks) and of undesirable redundancy which, altogether, improves the solution speed.

Future work. The objectives of the future work are to find more direct transformations for global constraints, investigating promising variable orderings to optimize the size of the DFAs and, similarly, variable and value orderings for the regular constraints. We wolud also research potential benefits from decomposition of automata, parallelization of the transformations and the solving process. Finally, we want to find out about general (static) criteria to decide when to apply the approach as well as extracting promising application areas in general.

References

1. Dechter, R.: Constraint processing. Elsevier Morgan Kaufmann (2003)
2. Hellsten, L., Pesant, G., van Beek, P.: A domain consistency algorithm for the stretch constraint. In: Wallace, M. (ed.) Principles and Practice of Constraint Programming - CP 2004. Lecture Notes in Computer Science, vol. 3258, pp. 290–304. Springer (2004)
3. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley (1979)
4. Marriott, K.: Programming with Constraints - An Introduction. MIT Press, Cambridge (1998)
5. Paltzer, N.: Regular Language Membership Constraint. Seminararbeit, Universität des Saarlandes, Deutschland (2008)
6. Pesant, G.: A filtering algorithm for the stretch constraint. In: Walsh, T. (ed.) Principles and Practice of Constraint Programming - CP 2001. Lecture Notes in Computer Science, vol. 2239, pp. 183–195. Springer (2001)

Robust Projectional Editing

Marcus Frenkel and Friedrich Steimann

FernUniversität in Hagen, Germany

Zusammenfassung Transaktionen stellen eine Möglichkeit dar, um in Code-Editoren eine dauerhafte semantische Korrektheit des Programmcodes zu gewährleisten, insbesondere, wenn sie durch Methoden wie die constraintbasierte Programmreparatur unterstützt werden. In ihrer bisherigen Erforschung sind sie allerdings lediglich für den lokalen Einsatz durch einen einzelnen Programmierer geeignet. Insbesondere beim kollaborativen Arbeiten auf einer gemeinsamen Codebasis entsteht das Problem, dass zwangsläufig Konflikte auftreten, wenn 2 oder mehr Programmierer die gleiche Stelle im Programmcode modifizieren wollen. Eine Lösung zum Beseitigen des Problems stellt die Verwendung des 2-Phase-Lockings dar, wobei die Herausforderung darin besteht, dieses vom (üblichen) Datenbank-Kontext auf die kollaborative Programmierung und das Setzen von Sperren auf Bestandteile des Programmcodes zu übertragen. In dem vorliegenden Kurzbericht diskutieren wir hierfür eine Lösungsmöglichkeit, welche insbesondere auf der Verwendung eines projizierenden Code-Editors aufbaut.

1 Ausgangslage

Projizierende Editoren (wie etwa MPS[1]) zeichnen sich dadurch aus, dass sie das Modell eines Programms zwar dem Programmierer in textueller Form präsentieren, intern jedoch alle Prozesse -- insbesondere auch das Hinzufügen, Verändern und Löschen von Programmelementen -- auf Basis des Programmmodells und entsprechend der Regeln des Metamodells der zugrundeliegenden Programmiersprache durchführen. Durch dieses Vorgehen ist zwar immer syntaktische Korrektheit gewährleistet, die Einhaltung semantischer Regeln kann hierdurch jedoch nicht erzwungen werden.

Eine Möglichkeit zum Gewährleisten semantischer Korrektheit in einem projizierenden Editor ist es, nur die Eingabe semantisch korrekter Werte beim Editieren einer Programmstelle zuzulassen. Eine Möglichkeit, dies sicherzustellen, ist über die Formulierung der semantischen Regeln mit Hilfe von formal notierten Invarianten (wie etwa in unseren Arbeiten zur constraintbasierten Programmreparatur[2] bereits erforscht wurde). Das Problem dieses Ansatzes ist, dass viele

[1] https://www.jetbrains.com/mps/

[2] Friedrich Steimann, Jörg Hagemann, and Bastian Ulke. 2016. Computing repair alternatives for malformed programs using constraint attribute grammars. In Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016. 711–730.

gewünschte Modifikationen an Programmcode nicht nur auf eine einzelne Codestelle beschränkt sind, sondern Änderungen an mehreren Stellen erfordern. Beispiele hierfür sind das Ändern des Typs einer Variablendeklaration mit anschließender Anpassung von Feldzugriffen und Methodenaufrufen auf dieser Variablen, das Verschieben eines Feldes in einen anderen Typ mit entsprechenden Anpassungen der Zugriffe auf dieses Feld oder das Hinzufügen einer neuen Methode in einen Typ im Kontext eines neuen Methodenaufrufes. Einige dieser Änderungen lassen sich in atomare Einzeloperationen aufsplitten, etwa das Hinzufügen einer neuen Methode (mit leerem oder Standard-Rumpf) und das anschließende Einfügen eines Aufrufes dieser Methoden; viele Änderungen erfordern jedoch einen temporär inkonsistenten Zustandes des Programms.

Eine Lösung dieses Problems ist das Zusammenfassen mehrerer Einzeländerungen in einer Transaktion, die während ihrer Durchführung inkonsistente Programmzustände erlaubt und die erst abgeschlossen werden kann, wenn nach Änderungen ein konsistenter Programmzustand hergestellt wurde. In einer vorherigen Veröffentlichung (siehe Fußnote 2) haben wir zu diesem Zweck den Solution Space Explorer (*SSE*) entwickelt, welcher dem Programmierer ausgehend von einem zu ändernden Attribut eines Programmelementes (etwa dem Typ einer Variablendeklaration) weitere Attribute von Programmelementen präsentiert, die mit Blick auf die ursprünglich gewünschte Änderung und mit Ziel der Herstellung von Konsistenz angepasst werden müssen.

Jede der durch den SSE durchgeführten Transaktionen erfüllt in einem solchen Fall das ACID-Prinzip. Die *atomicity* wird durch den Transaktionscharakter selber und die *consistency* über die semantische Prüfung mittels Invarianten sichergestellt. Werden Änderungen nur lokal durch einen einzelnen Programmierer durchgeführt, ist die *isolation* der Transaktionen ebenfalls erfüllt; die *durability* ergibt sich implizit aus dem Speichern des geänderten Programmcodes.

2 Problemstellung und Lösung

Der Solution Space Explorer funktioniert, solange lediglich ein einzelner Programmierer den Code lokal modifizieren möchte. Anders sieht es aus, wenn kollaborativ programmiert werden soll, also mehrere Programmierer gleichzeitig auf der gleichen Codebasis arbeiten. Prinzipiell erlaubt der Solution Space Explorer diesen Ansatz, und zwar indem er nach jeder abgeschlossenen Transaktion die Codeänderungen per Broadcast an alle Clients verteilt.

Ein Problem ergibt sich jedoch in den Fällen, in denen mehrere Programmierer gleichzeitig die gleiche Codestelle bearbeiten möchten. In einem solchen Fall kann das Prinzip der *isolation* verletzt werden, wenn miteinander in Konflikt stehende Änderungen durchgeführt werden sollen.

Ein verbreiteter Mechanismus zum Gewährleisten von *isolation* gleichzeitig durchgeführter Transaktionen ist das 2-Phase-Locking. Bei diesem Ansatz werden in einer Transaktion auf zu lesenden und zu ändernden Daten Read- beziehungsweise Write-Locks gesetzt, um konfligierende Änderungen zu verhindern. Um Reihenfolgeabhängigkeiten zu vermeiden, erfolgt die Lock-Behandlung in 2

Phasen: In der ersten werden durch eine Transaktion Locks ausschließlich gesetzt und keine freigegeben; nach Durchführung aller benötigten Änderungen werden in der zweiten Phase alle Locks wieder freigegeben, ohne neue zu setzen.

Übertragen auf den Solution Space Explorer würde dieser Read-Locks auf alle Attribute von Programmelementen setzen, die dem Programmierer zur Auswahl präsentiert werden. Entscheidet er sich für einen bestimmten Wert eines Attributs, wird vor Ausführen der Änderung ein Write-Lock auf das entsprechende Attribut gesetzt. Beim Erweitern der Auswahl der präsentierten Attribute für eine umfangreichere Lösung müssen dementsprechend weitere Read-Locks gesetzt werden. Wurde das Programm derart in einen (neuen) konsistenten Zustand überführt, können die Transaktion bestätigt und im Zuge dessen alle gesammelten Locks freigegeben werden. Eine Propagierung der neuen Werte per Broadcast an alle Clients ist erst nach Abschluss der Transaktion notwendig, da durch die gesetzten Write-Locks kein anderer Client auf die entsprechenden Attribute zugreifen könnte. Projizierende Editoren wie MPS stellen eine ideale Grundlage für die Implementierung eines derartigen Verhaltens dar, da sie im Gegensatz zu Texteditoren intern auf einem Syntaxbaum arbeiten, dessen Knoten sich durch IDs eindeutig identifizieren und damit sperren lassen; textbasierte Editoren müssten dagegen etwa zeilenweise sperren, was weit ungenauer -- und bei neu eingefügten Zeilen deutlich schwieriger -- umzusetzen wäre.

Der beschriebene Ansatz weist zwei Probleme auf, die für das 2-Phase-Locking üblich sind und die behandelt werden müssen: Deadlocks und eine hohe Anzahl von Locks, die das gleichzeitige Durchführen mehrerer Transaktionen verhindern. Im Falle des SSE entstehen beide Probleme dadurch, dass durch die Auswahl von weiteren Attributen von Programmelementen, welche das ursprünglich zu ändernde Attribut einschränken, sehr schnell sehr viele Read-Locks gesetzt werden müssen, die dementsprechend eine Parallelisierung von Transaktionen unterschiedlicher Clients verhindern oder zu Deadlocks führen. Wichtig ist daher bei der Umsetzung des Locking-Ansatzes im SSE eine *gezielte* und *sparsame* Auswahl von präsentierten Elementen, um die Anzahl der Locks niedrig zu halten.

Unit Testing von Datenbank-getriebenen Java Enterprise Edition Anwendungen

Andreas Fuchs

WWU Münster
andreas.fuchs@wi.uni-muenster.de

Abstract. Die Java Enterprise Edition (Java EE) Plattform und die dazugehörige Java Persistence API (JPA) sind weit verbreitete Technologien um Anwendungen zu entwickeln, die mit Datenbanken interagieren. Viele Kontrollflüsse von diesen Anwendungen hängen stark vom Zustand der angebundenen Datenbank ab. Testfälle, die das Ziel einer Kontrollflussabdeckung der zu testenden Anwendung haben, müssen daher für diese Art von Anwendungen sowohl Testdaten für die Eingabe der Anwendung erstellen, als auch die Datenbank in einen passenden Zustand vor Testfallausführung versetzen. Wir führen eine Java EE Anwendung auf Basis ihres Bytecodes symbolisch aus, um Constraints aus dem Kontrollfluss der Anwendung mit Datenbank-Constraints zu verbinden. Dies ermöglicht uns, für jeden möglichen Kontrollfluss der Anwendung Eingabedaten und benötigte JPA Entitätsobjekte für den Datenbankzustand zu erstellen. Wir haben unseren Ansatz in ein automatisches Testfall-Generierungswerkzeug eingebaut. Eine experimentelle Bewertung zeigt, dass wir dadurch die Kontrollflussabdeckung von Java EE Anwendungen erhöhen.

Keywords: Symbolische Ausführung, Automatische Testfallgenerierung, Java Enterprise Edition, Java Persistence API

DSI: Automated Detection of Dynamic Data Structures in C Programs and Binary Code*

Thomas Rupprecht, Jan H. Boockmann, David H. White and Gerald Lüttgen

Software Technologies Research Group, University of Bamberg, 96045 Germany

Abstract. Program comprehension is an important task for software engineers who maintain legacy code, as well as for reverse engineers who analyse binary executables such as malware. Detecting dynamic, i.e., pointer-based data structures is a particular challenge due to the complex usage of pointers found in real world software.
This paper presents the key results of the DFG-funded project "Learning Data Structure Behaviour from Executions of Pointer Programs" (DSI), in which dynamic analysis techniques have been developed to identify dynamic data structures in C programs and x86 binary code. DSI's analysis utilizes a novel memory abstraction that allows for a compact description of pointer-based data structures such as linked lists and binary trees, and their interconnections such as parent-child nesting. On top of this abstraction, an evidence-collecting approach calculates a natural language description of the observed data structure with the help of a systematic taxonomy. The inferred data structure information is not only helpful for program comprehension but also for other use cases including software verification and software visualization.

1 Introduction

This paper summarizes the key results of the DFG-funded project DSI (LU 1748/4-1), whose main goal was to develop novel techniques for identifying dynamic data structures in pointer software. Details can be found in [23, 24, 27, 28] and on the DSI webpage located at `http://www.swt-bamberg.de/dsi`.

Motivation. Many software developers are faced with legacy code, or sophisticated program constructs employed in low-level code. In addition, programs often rely on dynamic, i.e., pointer-based data structures involving linked lists and trees. Therefore, developers desire advanced support for program comprehension, which is even more true when only a binary version of a program is available, e.g., when analysing the vastly growing amount of malware [3]. To partially alleviate this obstacle, we propose *Data Structure Investigator* (DSI), a novel dynamic analysis for the automatic identification of dynamic data structures. DSI detects (cyclic) singly and doubly linked lists and binary trees, as

* This research is supported by the German Research Foundation (DFG) under project title "Learning Data Structure Behaviour from Executions of Pointer Programs" (grant no. LU 1748/4-1).

well as other data structures, such as skip lists, that are not handled by related work [8,14,17]. Additionally, DSI allows for arbitrary parent-child nesting combinations of such data structures, which is also out-of-scope of [8,12,14,15,17]. DSI's analysis information can be used in various ways, such as for inferring a natural language description of an observed data structure (e.g., "a doubly linked list with nested singly linked lists"), for generating verification conditions for verification tools such as VeriFast [16], and for enabling advanced visualizations of pointer programs.

State-of-the-art. Prominent examples of dynamic analysis tools for detecting dynamic data structures are ARTISTE [8], DDT [17], and MemPick [14]. These operate on binaries only and instrument the binaries to extract runtime information. DSI also uses instrumentation to record memory events, but operates on both source code and binary code. One of the biggest challenges when identifying data structures comes with data structure operations as they tend to temporarily break a data structure's shape; consider, e.g., the rewriting of pointers during a doubly-linked list insertion operation. We term the true shape of a data structure a *stable shape* and the temporarily broken shape a *degenerate shape*. Both DDT and MemPick avoid degenerate shapes: DDT performs its analysis on operation boundaries and thus requires their accurate detection, while MemPick employs a heuristic to determine quiescent periods during which no changes to the data structure are made. While DSI also analyses degenerate shapes, it uses an evidence-collecting approach so as not to loose overall precision. ARTISTE does not explicitly avoid degenerate shapes, too, but it becomes more conservative with its analysis when it encounters them. Additionally, ARTISTE, DDT, and MemPick have the common assumption that one node of a data structure occupies an allocated chunk of memory as a whole. Instead, DSI has a finer grained heap abstraction, which allows it to seamlessly deal with advanced and low-level constructs such as employed, e.g., by the Linux kernel list [2].

Complementary to dynamic analysis tools are static shape analysis tools such as Predator [12] and Forester [15], which operate on source code. They also infer the shape of a data structure but focus on program verification rather than program comprehension. Other related work are visualization tools that generate a compact view of the heap to ease understanding, e.g., Heapviz [5] and HeapDbg [19]. Both operate on heap snapshots as opposed to DSI's dynamic analysis that observes how data structures evolve over time. Heapviz processes Java-based heap snapshots that provide rich information about objects, their types, and their interconnections. It uses the extracted type information from the heap as is, which can result in less precise data structure naming, as can be seen in an example in [5], where a doubly linked list (DLL) is labeled as `LinkedList`. Instead, DSI infers shapes without relying on preexisting data structure information, and will correctly report a DLL in the mentioned example. HeapDbg is similar to Heapviz but infers shapes by inspecting the interconnections between objects; for example, it classifies objects with a left pointer and a right pointer as a tree. In addition, HeapDbg does not support as many data structures as DSI does, such as skip lists and various parent-child relations.

Contributions. DSI contributes to the state-of-the-art of program comprehension and reverse engineering with a novel approach for identifying dynamic data structures in both source code and binaries. Its dynamic analysis is based on a fine-grained memory abstraction that consists of singly linked lists and their interconnections (such as nesting), where list nodes can cover sub-regions of memory. The detectable data structures are summarized in a taxonomy and reach from (cyclic) singly and doubly linked lists to skip lists to trees. At the heart of DSI is an evidence-collecting algorithm called *DSIcore* (see Sec. 3), which considers the structural complexity of an observed data structure shape as evidence measure, and reinforces evidence by exploiting structural and temporal repetition to discriminate against degenerate shapes.

Two concrete realizations of DSI have been developed as front ends to DSIcore: (i) *DSIsrc* supports the analysis of C source code (see Sec. 4), and (ii) *DSIbin* supports x86 binaries (see Sec. 5). One of the fundamental differences between DSIsrc and DSIbin is the availability of type information in source code but not in (stripped) binaries. This is mitigated in DSIbin by first using the state-of-the-art type excavator Howard [25] to recover (partial) low-level type information, and subsequently refining the types via a new approach termed *DSIref*, which employs DSIcore to assist in the inference of low-level types via high-level data structure information (see Sec. 5).

Extensive benchmarking using example programs from the literature and real-world programs demonstrated that DSI identifies dynamic data structures correctly and robustly. Due to the generality of DSIcore's memory abstraction and in contrast to related work, DSI provides rich descriptions of data structures over a variety of implementation techniques that commonly occur in pointer programs. Three back ends to DSIcore have also been prototyped within the DSI project and testify to DSI's applicability in various domains, namely program verification, data structure visualization, and operation detection (see Sec. 6).

2 DSI Tool Suite

Fig. 1 depicts DSI's architecture: our front-end components DSIsrc and DSIbin, the DSIcore component and various back-end components. Both front ends collect runtime information via a dynamic analysis, which is accomplished by instrumenting C source code via the CIL framework [21] and binaries via Intel's Pin framework [18]. Executing an instrumented program results in an execution trace that records relevant heap and stack events such as memory (de-)allocations and pointer writes. The trace is passed to DSIcore for offline analysis. DSIbin uses Howard [25] for recovering type information from binaries and our additional type refinement component DSIref. DSI's back ends consume the information produced by DSIcore and comprise the *naming* module for giving a natural language description for an identified data structure, the *operation detection* component for localizing data structure operations, the *visualization* component for presenting the data structure consistently over its lifetime, and the *verification* component for interfacing to the program verifier VeriFast [16] (see Sec. 6).

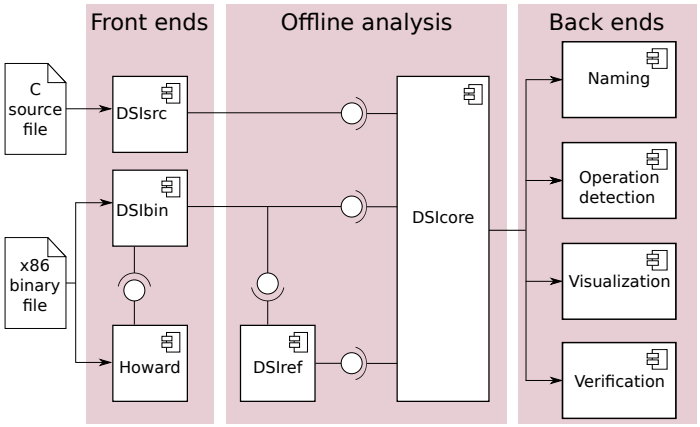


Fig. 1. Overview of the DSI tool chain. (Figure adapted from [23].)

3 DSIcore: DSI's Core Algorithm

This section overviews our general DSI approach implemented by DSIcore, using the illustrative example of Fig. 2. The example shows two time steps t and $t+1$ in the construction of a singly linked list (SLL) of doubly liked lists (DLLs), with entry pointer `ep`. Note that there exists a degenerate DLL child at time step t, and how the pipelined nature of the approach becomes apparent. The pipeline starts by constructing a sequence of *points-to graphs* that model the heap at each time step, where a points-to graph represents allocated memory chunks as vertices and pointers as edges (see Fig. 2(a)).

Memory abstraction. DSI relaxes the common assumption that data structure nodes occupy an allocated memory chunk as a whole [8, 14, 17]. Instead, our approach allows nodes to cover sub-regions of memory, termed *cells*. Sequences of cells that have the same linkage condition, i.e., all pointers originate at the same linkage offset relative to the cell's start address, and that point to the start address of the following cell are called *strands*. Strands are the basic data structure building blocks considered by DSI and highlighted in Fig. 2(a) by the thick coloured arrows that traverse through nodes and that are labeled S_1 to S_6. To infer a data structure shape, the relationships between strands need to be identified, which we term *strand connections* (SCs). A strand connection describes exactly one way in which multiple cells of a strand are related. It aggregates all *cell pairs* of two strands that have the exact same relation; thus, a strand connection consists of a set of cell pairs. Observe that two strands can be related in more than one way, resulting in multiple strand connections between the strands. We construct a *strand graph*, where vertices represent strands and edges represent strand connections (see Fig. 2(b)). The strand connections with the same edge style denote the same relationship type; for example, the DLL

strands form a tight strand connection that is bi-directional, e.g., S_3 can be reached from S_2 and vice-versa. Two kinds of loose and uni-directional strand connections are formed between the parent SLL and each child DLL; for example, S_1 cannot be reached from S_2.

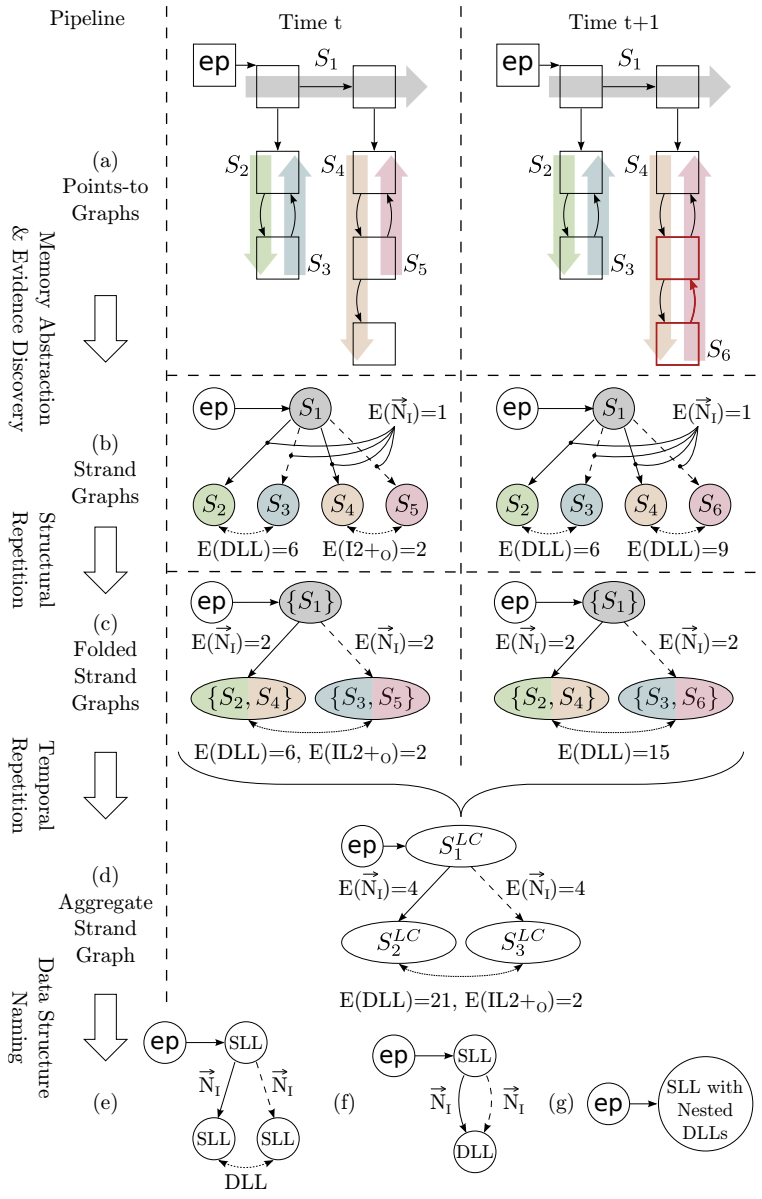


Fig. 2. Left: DSI's pipeline; right: the example of Sec. 3. (Figure adopted from [28].)

In the following, data structures and their interconnections are consolidated under the term *memory structure*. A memory structure observation is made on the strand graph by matching (parts of) the graph against a set of shape predicates. For example, our DLL predicate requires exactly two strands running in opposite directions and connected by a tight strand connection. The predicates available to DSIcore are defined by our data structure taxonomy presented in [28], which also describes the precedence rules on predicates when matching data structures; these are needed to avoid ambiguities and not inadvertently interpret, e.g., a binary tree as parent-child nesting of SLLs.

Evidence collection. Each memory structure observed in a strand graph is associated with a corresponding memory structure label and an evidence count E. The evidence is recorded on the strand connections that are matched by the shape predicate. The evidence count is derived from the structural complexity of the memory structure on the basis of the number of cell pairs comprising the strand connection and the way in which these are accessed by the shape predicate. Evidence weights for our example, are shown in Fig. 2(b). The degenerate DLL in time step t has an evidence count of 2 for "intersecting on two nodes overlay" ($\mathrm{I2}+_{\mathrm{O}}$), where the number of connections between the two strands is required to be at least two and the count is simply the number of cell pairs in the connection. The regained stable DLL shape at $t+1$ is matched by a predicate checking that the forward/reverse property of the DLL holds for each cell pair in the strand connection. This results in a count of 3 for each cell pair: 1 for the existence of the cell pair, plus 2 because both cells in the cell pair must be analysed. As three cell pairs exist, the total evidence count is $3 * 3 = 9$. Nesting ($\mathrm{N_I}$) only requires the existence of one cell pair connecting the parent strand to the child; hence, each occurrence has a count of 1. This evidence is spread over each strand connection that participates in the parent-child nesting and is then accumulated via *structural* and *temporal repetition*:

- Regarding structural repetition, DSI detects and groups those elements of the strand graph that perform the same role within one time step. Consider, e.g., the forward strands $\{S_2, S_4\}$ and backward strands $\{S_3, S_5\}$ of the DLL children at time t in Fig. 2. The grouping is done via a merge algorithm that produces a *folded strand graph* (see Fig. 2(c)). The folding results in vertices that now consist of strand *sets* and merged strand connections, where all evidence counts are summed up. The aggregation of the strand connections reinforces the evidence and is part of our solution to rule out degenerate shapes during data structure operations, as those parts of the data structure that are in stable shape can be aggregated with others that are in degenerate shape. In practice, the degenerate shape is in the minority, such that the majority of stable shapes overrides the minority.
- To track the temporal behavior of a memory structure and enable the identification of temporal repetition, it must be determined which strands represent the same data structure building block over multiple time steps. We tackle this problem by considering the labeling from the point of view of

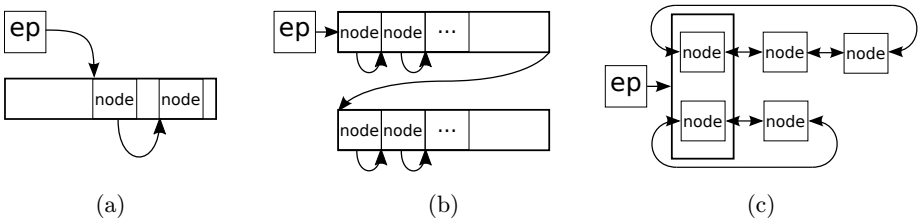


Fig. 3. Complications of C heaps: (a) custom allocator, (b) cache efficient list [7], (c) Linux kernel cyclic DLL [2]; `ep` denotes an entry pointer. (Figure adopted from [28].)

each entry pointer to a data structure separately, because entry pointers are inherently stable over their lifetimes. For each entry pointer and each time step in which an entry pointer exists, temporal repetition is performed by extracting the folded strand graph's subgraph reachable from the entry pointer. All extracted subgraphs are then merged over the lifetime of the entry pointer, resulting in an *aggregate strand graph*, in which the identified evidences are again accumulated. This is the second part of the evidence reinforcement to rule out degenerate shapes. The vertices of the aggregate strand graph are abstract representations of the strands in terms of their *linkage conditions* (S^{LC}).

Our example's aggregate strand graph is shown in Fig. 2(d), where the evidence for DLL is overwhelming. At the end of an entry pointer's lifetime, we interpret DSI's analysis by choosing the label with the highest evidence for each strand connection and label each strand as SLL or cyclic SLL (CSLL) (see Fig. 2(e)).

Naming of data structures. While the aggregate strand graph is useful for program comprehension, a natural language annotation of the program's source code at an entry pointer declaration, which names the data structure to which the entry pointer points, can be preferable. Of course, this assumes that the source code is available. Our according *naming* component iteratively groups the vertices of the aggregate strand graph and assigns a textual label to the resulting group; these grouped elements now form an atomic vertex in subsequent groupings. For example in Fig. 2(f), the result of grouping the two vertices connected by a DLL strand connection is shown. The order in which vertices are grouped must be carefully chosen so that the most suitable naming is generated; see [28] for details. Ultimately, we end up for our example with the graph in Fig. 2(g).

4 DSIsrc: DSI Front End for Analysing C Source Code

DSIsrc instruments the source code using the CIL framework [21], inserting instructions for recording pointer writes and memory (de-)allocations both on the heap and the stack. Subsequently, the instrumented source code is compiled

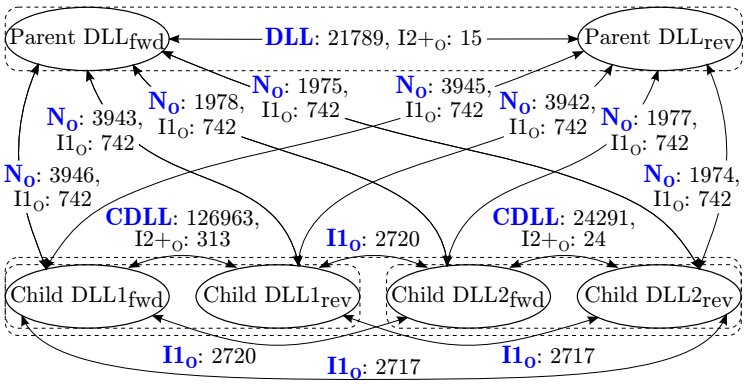


Fig. 4. Aggregate strand graph for `libusb`. (Figure adopted from [28].)

and executed to capture the programs event trace that is then analysed by DSIcore. The following paragraphs are adapted from [28], where a more detailed description of DSIsrc can be found.

The C heap. The reason for choosing C as a concrete realization for DSI is that it is far less restrictive regarding heap manipulations than other programming languages such as C# or Java. This results in programs that utilize the freedom of pointer arithmetic, type casts, macro usage, and (customized) memory allocations, and that are quite often favoring performance over source code readability. Together with C's widespread use in operating systems code and in Unix systems, the amount of difficult-to-comprehend legacy code is overwhelming, making DSI's capabilities a desirable feature for software developers.

A common assumption of related work is that one node of a data structure corresponds to one allocated chunk of memory. This assumption breaks in case multiple nodes of a data structure are placed inside a memory chunk, which is often the case with (i) custom memory allocators (see Fig. 3(a)), (ii) cache-efficient data structures [7] (see Fig. 3(b)), and (iii) head nodes of multiple lists embedded in the same memory chunk (see Fig. 3(c)). The latter is common practice in the cyclic Linux kernel list [2], which embeds the linkage struct `list_head` inside another struct. To model these situations, it is mandatory to allow nodes to only cover some sub-region of memory and permit that the linkage offsets are always from the start of the enclosing sub-region instead of, as is done in [14,17,19], the start of the enclosing memory chunk. DSI explicitly supports such heap states, which are typical for low-level C programs, with its cell-based strand abstraction.

Example. `libusb` [1] is one of the most challenging examples from the DSI benchmark [28]. This benchmark comprises textbook examples, self-written synthetic examples, Forester/Predator examples [4], and the real-world programs `bash` and `libusb`. The latter is a user space usb library, which comprises almost 7k lines

of C code. We exercised this code using the included utility `listdevs`, modified to expose several `struct libusb_context`s. The main data structure of `libusb` forms a parent CDLL, where both child elements for the parent form a CDLL: one for devices and one for associated file descriptors. DSI's output, as depicted in Fig. 4, clearly indicates this structure. Note that the CDLLs are Linux CDLLs that are embedded in structs, thus requiring our cell abstraction to understand the cyclic nature of the lists.

5 DSIbin: DSI Front End for Analysing x86 Binaries

DSIbin opens up our DSI approach to x86 binaries. The challenge here is not the binary format itself but the absence of any type information that is needed by DSIcore for detecting and tracking cells and strands.

Front end. To provide DSIcore with the required information when inspecting binaries, we utilize Intel's Pin framework [18] for capturing, as before, pointer writes and memory (de-)allocations. DSIcore demands low-level type information including types of (nested) structs both on the heap and stack, which is unavailable from stripped binaries. Therefore, DSIbin relies on the type excavator Howard [25] to recover this information. One fundamental problem when inferring types is to identify whether some types are the same. This happens in case one type is allocated at different locations within the binary. Because no explicit information is present in the binary that indicates type equivalence, this needs to be inferred separately. We modified Howard mildly to perform type merging between identified structs by tracking whether instructions and pointers touch binary compatible memory chunks from different allocation sites, which then get merged. However, some situations are not covered; for example, Howard does not merge nested types and misses nested structs when the access pattern is ambiguous, e.g., when a nested struct is placed at the head of the surrounding struct. To overcome these limitations, we devised the type refinement component DSIref.

DSIref: Refinement of type information. DSIref uses Howard's inferred types as starting point. Pointer connections reveal information about the data structure layout; for example, an incoming pointer to the middle of a data structure might indicate a nested struct or a linkage between two objects of different types. Struct types as reported by Howard could be merged when they are binary compatible, i.e., they have the same size and fields of the same primitive data types. Because pointer connections can be ambiguous, we create a set of possible type interpretations and then select the most plausible one. The latter is done by evaluating each interpretation with DSI and choosing the structurally most complex data structure as the most plausible interpretation. The intuition is that correct type merges and nested type detections naturally reveal the most 'complex' data structure. For example, when considering Fig. 6, the cyclicity of the lists can only be detected when the grey nodes are correctly identified.

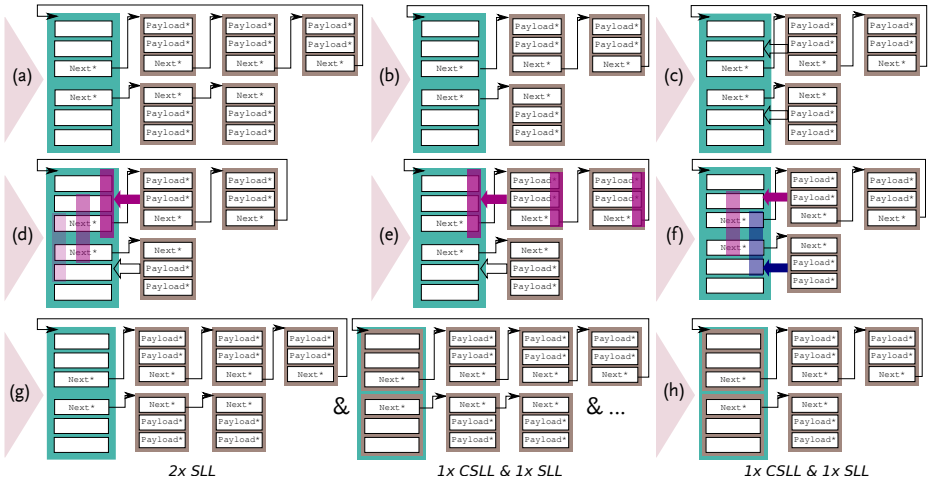


Fig. 5. Overview of the DSIref approach: (a) create sequence of points-to-graphs from program execution (only one shown); (b) construct merged type graph capturing pointer connections between types; (c) exploit pointer connections by mapping type sub-regions (two possibilities shown); (d) observe that multiple interpretations may be possible; (e) propagate each interpretation along pointer connections; (f) rule out inconsistencies; (g) evaluate remaining interpretations via DSI; (h) choose the 'best' interpretation in terms of data structure complexity (indicated by merged type graph with resulting label *1x CSLL & 1x SLL*). (Figure adopted from [23].)

The refinement process involves eight phases (Phases (a)–(h)), see Fig. 5. In Phase (a), DSIref takes all the 'as is' type information from Howard. In Phase (b), DSIref constructs a *merged type graph* similar to [5, 8, 22], where types are vertices and pointers are edges. Each type occurs only once, which reflects the connections between all types that occur in the execution trace under analysis. Notably, heap and stack allocated types are transparent to the merged type graph. This allows us to merge both, something that is not considered in related work [9]. Phases (c)–(f) utilize the merged type graph in order to generate new possible type interpretations. The first of these phases maps binary compatible sub-regions between different types by following pointer connections. Such mappings might be ambiguous as can be seen by the three possible mappings in Phase (d). Consequently, each of the mappings results in one type interpretation. In Phase (e), these types are then propagated along pointer chains as long as binary compatibility allows further propagation. This merges arbitrary combinations of (nested) struct types distributed over the heap and stack. Note that type interpretations can be conflicting as shown in Phase (f), where two interpretations are overlapping. These conflicts are eliminated as nested structs cannot overlap. All remaining interpretations and also Howard's initially inferred interpretation are evaluated with DSI, and the structurally most complex data

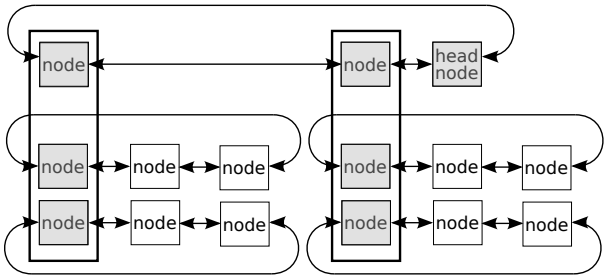


Fig. 6. CDLL parent with two CDLL children per parent node [4].

structure interpretation is selected. This is shown in Phase (h), where the cyclic SLL is chosen over the non-cyclic SLL interpretation.

Example. One challenging example from our benchmark for DSIbin [23] is taken from the literature [4] and happens to have the same dynamic data structure as `libusb` (see Fig. 4). A points-to graph of this example is shown in Fig. 6, where the parent CDLL is located at the top. It consists of two payload nodes, both having two nested CDLL children, and the external head node of the list. The nodes highlighted in grey are required to detect the correct shape of the data structure and are subject to the refinement performed by DSIref. Without DSIref, the parent nested structs and the external head struct are not detected to be of the same type, which results in missing the cyclic property of the DLL, i.e., the external head is cut off. The same is true for the grey head nodes of the child elements, with the addition that the overlay nesting is missed and only a "nesting on indirection" is reported. Therefore, without the refinement, a "DLL with two indirect nested DLL children" is detected. When employing DSIref, the grey nodes are also typed correctly, resulting in the identification of the data structure's true shape, i.e., a "CDLL with two overlay nested CDLL children."

6 DSI's Back Ends

The artefacts generated by DSIcore can be passed to various back-end components (see Fig. 1). Within the DSI project, we have developed early prototypes for operation detection, formal verification, and visualization. The *operation detection* component observes repetitive changes in data structures that occur due to insertion and deletion operations. The repetitive behaviour is identified via a multi-dimensional pattern matching approach driven by a genetic algorithm [11].

Detecting data structure operations is key for an important use case of DSI, namely the automated generation of source code annotations for formal program verification, such as pre-and post-conditions. The intention is to free a verification engineer from the burden of providing the often rather straightforward annotations that relate to (preserving) data structure shape. The utility

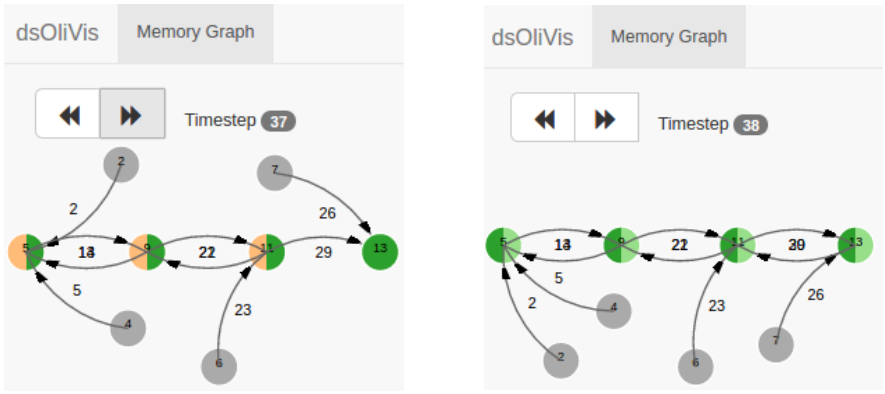


Fig. 7. Visualization of the transition of a DLL from a degenerate shape (left) to a stable shape (right). (Figure adopted from [26].)

of this DSI use case had already been demonstrated by interfacing DSI's predecessor tool dsOli [27] to the verification tool VeriFast [20], and DSI's verification component has been prototyped in [6].

Our third prototypical back end implements an advanced visualization approach [26] that uses the data structure information inferred by DSIcore to consistently display a data structure over its lifetime. This allows for a stepwise inspection even during periods of degenerate shapes, where our layout algorithm still arranges the data structure according to the final data structure interpretation. This is in contrast to, e.g., the *dot* renderer of Graphviz [13] which might rearrange the graph from one time step to the next. An example of adequately visualizing a DLL during an insertion operation is shown in Fig. 7, where grey nodes represent the entry pointers into the DLL. Observe that all DLL nodes, i.e., the nodes 5, 9, 11, and 13, are displayed consistently across both time steps.

7 Conclusions

We provided an overview of the novel DSI approach for reliably identifying dynamic data structures in pointer programs, which we envisage to be a significant help when comprehending C source code or reverse engineering binaries of pointer programs. The data structures in scope for DSI contain list-based structures such as (cyclic) singly and doubly linked lists (with head and/or tail pointers), binary trees, skip lists, and more complex data structures that are build from the mentioned ones via indirect and overlay nesting.

DSI advances over related work in terms of accuracy and data structure variety have been enabled by DSIcore's fine-grained memory abstraction in terms of cells, strands, and strand connections, and by DSI's evidence-collecting approach to data structure identification. In addition, we compensated the loss of type information in binaries via a novel combination of the type excavator Howard with DSIcore, whereby the high-level data structure information obtained by DSIcore

is employed to infer low-level type information such as nested struct types. We refer the interested reader to our publications on DSIsrc [28] and DSIbin [23,24] for more details on the DSI approach and its evaluation. DSI is open source and available for download from `http://www.swt-bamberg.de/dsi`.

The present paper also briefly discussed our proof-of-concept back end implementations, demonstrating how DSI's analysis results can be used for operation detection [11], verification condition generation [6], and data structure visualization [26]. Regarding additional future uses cases for DSI, we envision to employ DSI for generating malware signatures so as to detect polymorphic malware families. This use case is inspired by the virus scanner Laika [10], which detects structure on pointer connections but does not identify the data structures themselves. Finally, we wish to investigate how far DSI's rich analysis can contribute to memory leak detection. DSIcore already performs its own memory leak detection, allowing it to exactly record where and when a reference to memory is lost. By integrating this information into DSI's data structure visualization, a powerful tool could emerge that allows software developers to replay the steps leading to a memory leak and thus to find the leak's cause.

Acknowledgments. We thank the German Research Foundation (DFG) for their support of the DSI project (DFG grant LU 1748/4-1), and also Linus Dietz and Kathrin Welzel who contributed to the project in the context of their bachelor and master theses.

References

1. libusb 1.0.20. `http://www.libusb.info/`. Accessed: 8th May 2017.
2. Linux kernel 4.1 cyclic DLL (`include/linux/list.h`). `http://www.kernel.org/`. Accessed: 31 August 2015.
3. Malware statistics by AV-TEST. `https://www.av-test.org/en/statistics/malware/`. Accessed: 16th June 2017.
4. Predator/Forester GIT repository. `https://github.com/kdudka/predator`. Accessed: 8th May 2017.
5. Aftandilian, E. E., Kelley, S., Gramazio, C., Ricci, N., Su, S. L., and Guyer, S. Z. Heapviz: Interactive heap visualization for program understanding and debugging. In *SOFTVIS '10*, pp. 53–62. ACM, 2010.
6. Boockmann, J. Automatic generation of data structure annotations for pointer program verification. Bachelor thesis, U. Bamberg, Germany, October 2016.
7. Braginsky, A. and Petrank, E. Locality-conscious lock-free linked lists. In *ICDCN '11*, vol. 6522 of *LNCS*, pp. 107–118. Springer, 2011.
8. Caballero, J., Grieco, G., Marron, M., Lin, Z., and Urbina, D. Artiste: Automatic generation of hybrid data structure signatures from binary code executions. Tech. Report TR-IMDEA-SW-2012-001, IMDEA Software Institute, Spain, 2012.
9. Caballero, J. and Lin, Z. Type inference on executables. *ACM Computing Surveys*, 48(4):1–65, 2016.
10. Cozzie, A., Stratton, F., Xue, H., and King, S. Digging for data structures. In *OSDI '08*, pp. 255–266. USENIX Association, 2008.
11. Dietz, L. Multidimensional repetitive pattern discovery for locating data structure operations. Bachelor thesis, U. Bamberg, Germany, April 2015.

12. Dudka, K., Peringer, P., and Vojnar, T. Byte-precise verification of low-level list manipulation. In *SAS '13*, vol. 7935 of *LNCS*, pp. 215–237. Springer, 2013.
13. Gansner, E. R. and North, S. C. An open graph visualization system and its applications to software engineering. *Software-Practice and Experience*, 30(11):1203–1233, 2000.
14. Haller, I., Slowinska, A., and Bos, H. Scalable data structure detection and classification for C/C++ binaries. *Empirical Software Engineering*, 21(3):778–810, 2016.
15. Holík, L., Lengál, O., Rogalewicz, A., Šimáček, J., and Vojnar, T. Fully automated shape analysis based on forest automata. In *CAV '13*, vol. 8044 of *LNCS*, pp. 740–755. Springer, 2013.
16. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., W. Penninckx, W., and Piessens, F. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *NFM '11*, vol. 6617 of *LNCS*, pp. 41–55. Springer, 2011.
17. Jung, C. and Clark, N. DDT: Design and evaluation of a dynamic program analysis for optimizing data structure usage. In *MICRO '09*, pp. 56–66. ACM, 2009.
18. Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. Pin: Building customized program analysis tools with dynamic instrumentation. *SIGPLAN Notices*, 40(6):190–200, 2005.
19. Marron, M., Sanchez, C., Su, Z., and Fähndrich, M. Abstracting runtime heaps for program understanding. *IEEE Transactions on Software Engineering*, 39(6):774–786, 2013.
20. Mühlberg, J. T., White, D. H., Dodds, M., Lüttgen, G., and Piessens, F. Learning assertions to verify linked-list programs. In *SEMF '15*, pp. 37–52. Springer, 2015.
21. Necula, G. C., McPeak, S., Rahul, S. P., and Weimer, W. Cil: Intermediate language and tools for analysis and transformation of c programs. In *CC '02*, vol. 2304 of *LNCS*, pp. 213–228. Springer, 2002.
22. Raman, E. and August, D. I. Recursive data structure profiling. In *Workshop on Memory System Performance*, pp. 5–14. ACM, 2005.
23. Rupprecht, T., Chen, X., Boockmann, J. H., Lüttgen, G., and Bos, H. DSIbin: Identifying dynamic data structures in C/C++ binaries. In *ASE '17*. IEEE, 2017. Accepted for publication.
24. Rupprecht, T., Chen, X., White, D. H., Mühlberg, J. T., Bos, H., and Lüttgen, G. POSTER: Identifying dynamic data structures in malware. In *CCS '16*, pp. 1772–1774. ACM, 2016.
25. Slowinska, A., Stancescu, T., and Bos, H. Howard: A dynamic excavator for reverse engineering data structures. In *NDSS '11*. The Internet Society, 2011.
26. Welzel, K. Heap visualisation using interactive memory graphs. Bachelor thesis, U. Bamberg, Germany, March 2016.
27. White, D. H., Rupprecht, T., and Lüttgen, G. dsOli2: Discovery and comprehension of interconnected lists in C programs. In 18th Coll. on Programming Languages and Foundations of Programming (Kolloquium Programmiersprachen, KPS '15), 2015. Proceedings available online at `http://www.complang.tuwien.ac.at/kps2015/proceedings`.
28. White, D. H., Rupprecht, T., and Lüttgen, G. DSI: An evidence-based approach to identify dynamic data structures in C programs. In *ISSTA '16*, pp. 259–269. ACM, 2016.

Autorenverzeichnis