In *Proceedings der Tagung Software Engineering 2005, Essen, Germany, 2005, Lecture Notes in Informatics.*

Logische und softwaretechnische Herausforderungen bei der Verifikation optimierender Compiler

Sabine Glesner und Jan Olaf Blech

Institut für Programmstrukturen und Datenorganisation
Universität Karlsruhe, 76128 Karlsruhe, Germany

Abstract: Korrektheit von Compilern ist notwendige Voraussetzung für die Korrektheit der damit übersetzten Software. Insbesondere optimierende Compiler sind oft fehlerhaft. In diesem Papier stellen wir nach einem Überblick über den Stand der Forschung unsere neuen Arbeiten zur Verifikation optimierender Compiler vor. Dabei diskutieren wir zum einen, welche logischen Probleme sich bei der formalen Verifikation von Übersetzungsalgorithmen in Compilern mittels Theorembeweisern ergeben und welche Lösungen wir dafür entwickelt haben. Zum anderen zeigen wir, wie man die Korrektheit auch realer optimierender Compiler mit beträchtlichem Implementierungsumfang sicherstellen kann. Damit tragen unsere Ergebnisse zur Korrektheit von Compilern, einem wichtigen Werkzeug in der Softwaretechnik, bei. Außerdem entwickeln wir auf diese Weise Methoden, die auch in anderen Anwendungsbereichen zur Verifikation von Software eingesetzt werden können.

1 Einleitung

Korrektheit von Compilern ist notwendig, um die Korrektheit damit übersetzter Software sicherzustellen. Auch wenn man im allgemeinen Compilern vertraut, haben diese Software-Werkzeuge dennoch ihre Fehler, wie die Fehlermeldungen gängiger Compiler [Bo02] oder z.B. der in [Ne01] diskutierte, besonders augenfällige Compilerfehler demonstrieren. Wer kennt nicht das Phänomen, dass man verzweifelt Fehler im eigenen Programm sucht und diese Fehler verschwinden, sobald man Optimierungsstufen ausschaltet?

In unseren Arbeiten lösen wir die Frage, wie optimierende Compiler, die sich als besonders fehleranfällig herausstellen, verifiziert werden können. Dabei stellen wir folgende Kriterien an unsere Lösung: Korrektheit soll formal, d.h. strikt in einem logischen System mittels eines maschinellen Theorembeweisers nachgewiesen werden. Außerdem sollen unsere Methoden für realistische Compiler anwendbar sein. Aus Forschungssicht ist dieses Problem aus drei Gründen interessant: Compiler sind relativ große Software-Systeme, so dass man daran erkennen kann, wie gut die eingesetzten und entwickelten Verifikationsmethoden skalieren. Auch aus semantisch-logischer Sicht ist dieses Problem interessant, weil nicht nur verfeinernde, sondern insbesondere auch strukturverändernde, optimierende Transformationen betrachtet werden, die mit bislang üblichen Verifikationsmethoden nicht behandelt werden können. Und schließlich sind viele der im Bereich der Verifikation von

Compilern entwickelten Methoden in anderen Software- und Hardwarebereichen einsetzbar. In diesem Papier geben wir nach einer Zusammenfassung des Stands der Forschung einen Überblick über unsere neuen Arbeiten zur Verifikation optimierender Übersetzer.

2 Korrektheit von Compilern: Stand von Forschung und Technik

Bei der Korrektheit von Übersetzern unterscheidet man zwei verschiedene Fragestellungen: Zum einen untersucht man, ob ein gegebener Übersetzungsalgorithmus korrekt ist, d.h. ob er die Bedeutung, die *Semantik* der transformierten Programme erhält. Zum anderen stellt man die Frage, ob ein eventuell zuvor verifizierter Übersetzungsalgorithmus in einem vorliegenden Compiler auch korrekt implementiert ist. Den ersten Korrektheitsbegriff, der sich auf die semantische Korrektheit der Übersetzungsalgorithmen bezieht, bezeichnet man mit *Übersetzungskorrektheit*, den zweiten, der die Korrektheit der Implementierungen in Compilern betrachtet, mit *Implementierungskorrektheit*. Implementierungskorrektheit wurde erstmalig in [Po81, CM86] betrachtet. Im folgenden stellen wir den Stand von Forschung und Technik bezüglich dieser beiden Begriffe dar.

Übersetzungskorrektheit: In der Literatur werden meist Verifikationen von Verfeinerungstransformationen betrachtet. Das sind solche Übersetzungen, bei denen die Struktur der Programme nicht verändert wird, sondern lediglich während der Übersetzung immer genauer festgelegt wird, wie die einzelnen Berechnungen auszuführen sind. Z.B. würde man bei der Übersetzung höherer Programmiersprachen in Maschinencode bestimmen, wie komplexe Datenstrukturen in die Speicherhierarchie des Prozessors abgebildet werden. Das bedeutet insbesondere, dass Programme nach einem *Divide et Impera*-Prinzip lokal übersetzt und anschließend die Übersetzungen wieder zusammengefügt werden können. Die entsprechenden Korrektheitsbeweise folgen diesem Prinzip: Korrektheit kann lokal gezeigt werden, und globale Korrektheit folgt daraus. Verifikationen nach diesem Schema finden sich u.a. in [DvHG03, SA97].

Implementierungskorrektheit: Die Frage der Implementierungskorrektheit von Compilern ist aus softwaretechnischer Sicht von großer Bedeutung, insbesondere wenn man bedenkt, dass Übersetzer heutzutage nicht von Hand geschrieben, sondern aus geeigneten Spezifikationen mittels Generatoren erzeugt werden. Man könnte nun versuchen, diese Generatoren selbst zu verifizieren. Angesichts der Größe dieser Systeme entfällt diese Option, insbesondere wenn man maschinelle Beweise in Theorembeweisern führen möchte. Mit heute verfügbaren Verifikationsmethoden kann man Software dieser Größenordnung (noch) nicht in den Griff bekommen. Man könnte stattdessen versuchen, die generierten Compiler zu verifizieren. Auch diese Möglichkeit entfällt, mit derselben Begründung.

Als Ausweg aus diesem Dilemma hat sich in den letzten Jahren Programmprüfung als die Methode der Wahl etabliert, in der Literatur auch als *translation validation* [PSS98] oder *program checking* [GGZ98] bekannt. Anstatt den Übersetzer zu verifizieren, verifiziert man nur sein Ergebnis. Dabei geht man so vor, dass man den Compiler mit einem

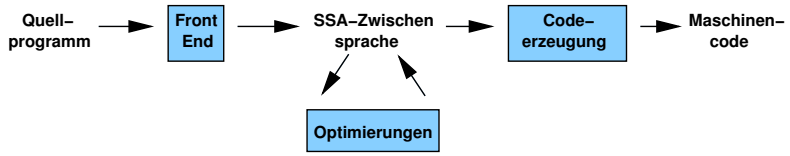


Abbildung 1: Architektur von Compilern

unabhängigen Checker anreichert. Der Checker erhält jeweils das Quellprogramm und das daraus generierte Zielprogramm als Eingabe und überprüft, ob die beiden Programme semantisch äquivalent sind. Im Falle einer positiven Entscheidung haben wir einen Nachweis, dass tatsächlich eine korrekte Übersetzung stattgefunden hat. Im negativen Fall weiß man gar nichts. Aus theoretischer Sicht kann man nicht erwarten, dass der Checker immer entscheiden kann, ob eine korrekte Transformation stattgefunden hat, weil Programmäquivalenz ein unentscheidbares Problem ist. Aus praktischer Sicht hat sich dieses Vorgehen aber als sehr gut geeignet erwiesen. Wenn man weiterhin den Checker bzgl. einer geeigneten Spezifikation formal verifiziert, dann erhält man im Fall einer positiven Entscheidung des Checkers ein formal verifiziertes Übersetzungsergebnis. Diese Methode zum Nachweis der Implementierungskorrektheit hat sich in allen Phasen der Frontends von Compilern (lexikalische, syntaktische und semantische Analyse) als sehr gut geeignet erwiesen, insbesondere auch deshalb, weil Ergebnisse in diesen Phasen der Übersetzung eindeutig sind [HGG+99, Gl03b, GFJ04]. Bei der syntaktischen Analyse z.B. muss man erstens prüfen, ob der vom Compiler berechnete Syntaxbaum tatsächlich zu der kontextfreien Grammatik der Programmiersprache passt, was in einem Top-Down-Durchlauf durch den Syntaxbaum erledigt werden kann, indem für jeden Knoten X_0 und seine Nachfolgerknoten $X_1, \ldots, X_n$ geprüft wird, ob es eine Produktion $X_0 ::= X_1 \cdots X_n$ gibt. Zweitens muss man sicherstellen, dass der Syntaxbaum eine korrekte Ableitung des ursprünglichen Programms ist, was man dadurch prüft, dass der Text, der durch die Aneinanderkettung der Blätter im Syntaxbaum entsteht, mit dem ursprünglichen Programm übereinstimmt.

Bisher nicht gelöste Probleme: Bislang wurde nicht geklärt, wie die Übersetzungskorrektheit strukturverändernder Transformationen nachgewiesen werden kann. Dieses Problem behandeln wir in Abschnitt 3. Dazu betrachten wir die Codeerzeugung aus SSA(*static single assignment*)-Zwischensprachen und diskutieren zwei alternative Beweismöglichkeiten. Bisherige Arbeiten haben außerdem nicht geklärt, wie für optimierende Transformationen, bei denen es mehrere, evtl. sogar viele korrekte Lösungen gibt, Implementierungskorrektheit sichergestellt werden kann. In Abschnitt 4 stellen wir eine Lösung dafür vor.

3 Übersetzungskorrektheit optimierender Compiler

In diesem Abschnitt betrachten wir die Codeerzeugung in Compilern und zeigen, wie für diese Phase die Übersetzungskorrektheit maschinell mit Hilfe eines Theorembewei-

sers nachgewiesen werden kann. Abbildung 1 zeigt die grobe Architektur von Compilern, die zuerst ein Quellprogramm mit dem Frontend (bestehend aus lexikalischer, syntaktischer und semantischer Analyse) in eine interne Zwischenrepräsentation transformieren. Diese Zwischenrepräsentation kann dann mit maschinenunabhängigen Transformationen optimiert werden. Anschließend wird in der Codeerzeugung ausführbarer Maschinencode erzeugt. Wir gehen von einer SSA(*static single assignment*)-basierten Zwischenrepräsentation [CFR+91] aus, weil damit die essentiellen Datenabhängigkeiten in Programmen direkt dargestellt werden und Optimierungen besonders gut möglich sind. Wir führen zunächst in Abschnitt 3.1 SSA-Zwischensprachen ein und stellen anschließend in den Abschnitten 3.2 und 3.3 zwei Beweismöglichkeiten für die Korrektheit der Codeerzeugung vor. Die Vor- und Nachteile der beiden Alternativen diskutieren wir in Abschnitt 3.4. In unseren Arbeiten haben wir den generischen Theorembeweiser Isabelle [NPW02] verwendet, der mit verschiedenen Logiken instantiiert werden kann. Wir verwenden die HOL(higher order logic)-Instantiierung, weil diese sich bereits bei der Formalisierung einer signifikanten Teilmenge von Java [KN03] bewährt hat.

3.1 SSA-Zwischensprachen

Wie die meisten Zwischensprachen in Compilern sind auch SSA-Sprachen[1] grundblockorientiert, d.h. maximale Sequenzen verzweigungsfreier Anweisungen werden in Grundblöcken angeordnet, und der Steuerfluss verbindet die Grundblöcke miteinander. Zudem wird in SSA-Darstellung jeder Variablen statisch nur einmal ein Wert zugewiesen, eine Darstellung, die man u.a. durch geeignete Duplizierung und Umbenennung der Variablen immer erreichen kann und wodurch man die essentiellen Datenabhängigkeiten eines Programms besonders gut erkennen kann. Innerhalb von Grundblöcken sind Berechnungen ausschließlich datenfluss-getrieben, d.h. eine Operation kann ausgeführt werden, sobald ihre Eingabewerte festliegen. In diesem Papier betrachten wir aus Platzgründen nur den (semantisch interessanteren) Fall der Compilierung von Basisblöcken und vernachlässigen alle Details, die für diesen Zweck unwichtig sind. Für eine detaillierte Darstellung verweisen wir auf [BG04]. Im Rahmen dieses Papiers kann man sich Basisblöcke als azyklische Datenflussgraphen (abgekürzt DAGs)[2] vorstellen.

3.2 Nachweis der Übersetzungskorrektheit

Grundlage jeder formalen Verifikation von Übersetzungen sind formale Semantiken der beteiligten Programmiersprachen. Wir benötigen also zunächst eine formale Semantik der SSA-Grundblöcke. Wie bereits oben erwähnt, kann man sich SSA-Grundblöcke als DAGs vorstellen. Die Frage, wie man DAGs bzw. Graphen im allgemeinen in einer logischen Sprache wie HOL darstellt, ist in der Forschung bislang nicht eindeutig geklärt und hängt von dem Kontext der Verifikationen ab. Wir haben uns dazu entschieden, *Termgraphen* als

[1] deutsch: SSA = Sprachen mit statischer Einmalzuweisung
[2] englisch: DAGs = directed acyclic graphs

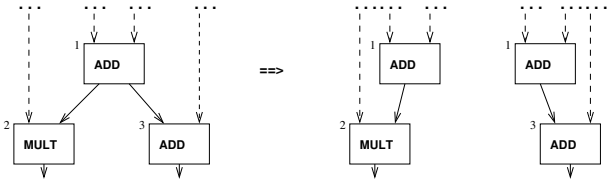


Abbildung 2: Transformation von SSA-DAGs in SSA-Bäume

Repräsentation zu wählen.

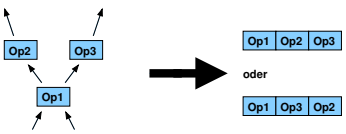
SSA-Grundblöcke enthalten gemeinsame Teilausdrücke, deren Ergebnisse an mehreren Stellen verwendet werden, nur einmal. Das ist der Grund dafür, dass SSA-Grundblöcke im allgemeinen keine Terme, sondern DAGs sind. Wir formalisieren SSA-Grundblöcke, indem wir ihre DAGs in eine äquivalente Menge von Termen umformen und dabei gemeinsame Teilausdrücke duplizieren, vgl. Abbildung 2. Um äquivalente Teilterme miteinander zu identifizieren, weisen wir jeder Operation im ursprünglichen SSA-DAG eine eindeutige Identifikationsnummer zu und duplizieren diese Nummer, wann immer wir einen Teilterm im DAG duplizieren. Auf diese Weise können wir einen SSA-DAG in eine Menge von SSA-Termen transformieren. In Isabelle/HOL haben wir derartige SSA-Terme mit folgender, hier nur vereinfacht dargestellter Definition spezifiziert:

datatype $SSATree = CONST\ value\ ident \mid NODE\ operator\ SSATree\ SSATree\ value\ ident$

Ohne auf die Isabelle-spzifischen Einzelheiten einzugehen, erkennt man, dass ein induktiver Datentyp $SSATree$ spezifiziert wird, dessen Elemente entweder aus einem Blatt bestehen oder aus zwei Teilbäumen zusammengesetzte Bäume sind. Weiterhin haben wir eine Funktion $eval\_tree$ definiert, die einen SSA-Baum nimmt und ihn auswertet, indem jedem Knoten, d.h. Operation im Baum ihr Ergebnis zugewiesen wird. Die Signatur dieser Funktion ist folgendermaßen definiert:

consts $eval\_tree :: ''SSATree \Rightarrow SSATree''$

Um nun nachzuweisen, dass die Erzeugung von Maschinencode korrekt ist, müssen wir die Semantik der Ziel-Maschinensprache wie auch die Abbildung der SSA-Grundblöcke in diese Sprache formalisieren. Wir haben eine relativ einfache Maschinensprache gewählt, deren Operationen direkt den entsprechenden Operationen der SSA-Grundblöcke entsprechen und deren Programme eine sequentielle Liste von Maschinenbefehlen sind, die der Reihe nach abgearbeitet werden. Für eine exakte Definition der entsprechenden Semantik verweisen wir auf [BG04]. Bei der Abbildung von SSA-DAGs in Maschinencode hat man einige Freiheitsgrade. Zum Beispiel muss man bei der Transformation des nebenstehend abgebildeten DAG zuerst die Operation op1 berechnen und hat dann die Wahl, ob zuerst op2 und dann op3 oder aber zuerst op3 und dann op2 berechnet werden soll. Wir konnten in Isabelle/HOL nachweisen, dass



jede Reihenfolge, die eine topologische Sortierung des SSA-DAGs ist, eine gültige Codeerzeugungsreihenfolge darstellt.

Mit diesem Beweis haben wir die Korrektheit der Abbildung zwischen den datenflussgetriebenen Berechnungen in SSA-DAGs und den sequentiell geordneten Befehlen im Maschinencode gezeigt. Diese Abbildung ist genau dann korrekt, wenn die Datenabhängigkeiten des SSA-Grundblocks erhalten bleiben. Für diesen Nachweis haben wir in Isabelle/HOL 885 Zeilen Beweiscode benötigt, was relativ umfangreich ist, insbesondere wenn man bedenkt, dass Beweise in Isabelle/HOL interaktiv von Hand geführt werden müssen und nicht generiert werden können. Außerdem erschien uns der Nachweis an einigen Stellen unnatürlich aufwändig. Diese Schwierigkeiten entstanden, weil wir unterschiedliche Induktionsprinzipien im SSA-Baum und in der Maschinencodeliste hatten (Induktion über Bäume sowie Induktion über Listen). Wir haben diese Beobachtungen zum Anlass genommen, einen völlig neuen Beweis zu erstellen, von dem wir im folgenden berichten.

3.3 Alternativer Nachweis der Übersetzungskorrektheit

Aufgrund der Beobachtung, dass Codeerzeugung aus SSA-Grundblöcken genau dann korrekt ist, wenn die Datenabhängigkeiten erhalten bleiben, haben wir eine völlig neue Spezifikation von SSA-Grundblöcken erstellt. Dabei haben wir SSA-Grundblöcke konsequent als partielle Ordnungen auf den in ihnen enthaltenen Operationen aufgefasst. Eine Operation o ist kleiner als eine andere Operation o', wenn o vor o' ausgewertet werden muss, weil das Ergebnis von o' direkt oder indirekt von dem Ergebnis von o abhängt. Wie in der mathematischen Logik üblich, haben wir eine partielle Ordnung O als eine Menge von Tupeln (x, y) repräsentiert, so dass aus $(x, y) \in O$ und $(y, z) \in O$ folgt, dass auch $(x, z) \in O$ (Transitivität) und dass aus $(x, y) \in O$ und $(y, x) \in O$ folgt, dass $x = y$ gilt (Antisymmetrie). Codeerzeugung haben wir als einen Prozess formalisiert, der weitere Abhängigkeiten in die partielle Ordnung einschleust und damit aus der partiellen eine totale Ordnung macht. Wir konnten damit nachweisen, dass Codeerzeugung korrekt ist, wenn die ursprüngliche SSA-Ordnung in der Ordnung des Maschinencodes enthalten ist. Für Einzelheiten des entsprechenden Isabelle-Beweises, für die uns hier leider der Platz fehlt, verweisen wir auf [BGLM04].

3.4 Diskussion der beiden Alternativen

Die in den Abschnitten 3.2 und 3.3 diskutierten Korrektheitsbeweise für Codeerzeugung aus SSA-Grundblöcken sind beide in dem Theorembeweiser Isabelle/HOL geführt worden und zeigen damit beide dasselbe Resultat, dass Codeerzeugung korrekt ist, wenn die Datenabhängigkeiten der SSA-Grundblöcke erhalten bleiben. Ist man nur an diesem Resultat interessiert, sind beide Beweise gleichwertig, insbesondere auch deshalb, weil sie sich in ihrer Länge kaum unterscheiden. Aus mathematisch-logischer Sicht unterscheiden sie sich jedoch stark. Dahinter steckt eine Erfahrung, die auch Mathematiker bei ihrer

Arbeit machen. Es gibt Beweise, die sich “gut” anfühlen, bei denen man die richtige Intuition getroffen hat. Man kann zwar keine formale Definition angeben, wann sich ein gegebener Beweis tatsächlich gut anfühlt, meist aber weiß man es genau, sobald man ihn hat, vgl. dazu auch [AZ04]. Diese Erfahrung haben wir auch gemacht. Der zweite Beweis basierend auf partiellen Ordnungen fühlt sich gut an. Die einzelnen Beweisschritte passen mit unserer intuitiven Beweisidee zusammen und formalisieren ein generelles Prinzip, nämlich dass die Transformation eines Programms die Datenabhängigkeiten erhalten muss. Dadurch, dass wir unseren Beweis auf dem generellen Prinzip “Erhaltung der Datenabhängigkeiten” aufgebaut haben, können wir den Beweis auch bei der Verifikation weiterer Transformationen wiederverwenden. In aktuellen Arbeiten verwenden wir ihn z.B. bei der Verifikation der Eliminierung toten Codes und bei der Verifikation von Schleifentransformationen, beides typische Optimierungen in modernen Compilern. Viele weitere Anwendungsmöglichkeiten existieren.

Formale Softwareverifikation mit Hilfe von Theorembeweisern wie Isabelle/HOL ist heutzutage aus zwei Gründen möglich: Erstens hat sich die Geschwindigkeit von Prozessoren so gesteigert, dass die Suchräume von Theorembeweisern ausreichend schnell durchlaufen werden können. Zweitens hat sich die Benutzerfreundlichkeit von Theorembeweisern so verbessert, dass diese Systeme auch von Arbeitsgruppen verwendet werden, die nicht an ihrer Entwicklung beteiligt waren. Trotzdem ist formale Softwareverifikation sehr teuer und erfordert aufwändige Benutzerinteraktionen. Dieser Aufwand kann reduziert werden, wenn es uns gelingt, Beweise wiederzuverwenden, indem wir Beweise auf allgemeingültigen Prinzipien für die Korrektheit von Systemtransformationen (wie hier die Erhaltung der Datenabhängigkeiten) aufbauen. Unsere hier vorgestellten Arbeiten zur Übersetzungskorrektheit sind dafür ein Beispiel.

4 Implementierungskorrektheit optimierender Compiler

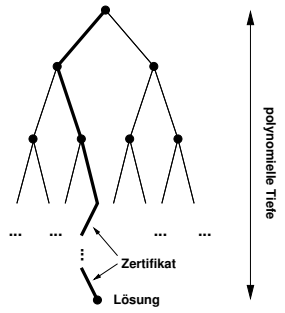


Abbildung 3: **NP**-Berechnungen

Bei der Codeerzeugungsphase ist wie bei fast allen Problemen in Backends von Compilern die Optimierungsvariante eines **NP**-vollständigen Problems zu lösen. Probleme in **NP** sind dadurch gekennzeichnet, dass ihre Lösungen immer in polynomieller Zeit auf Korrektheit geprüft werden können [Pa94]. Das bedeutet, dass eine nicht-deterministische Turingmaschine zwar einen potentiell exponentiell großen Suchraum durchlaufen muss, bevor sie eine Lösung findet, die Lösung selbst ist aber immer in polynomieller Tiefe zu finden. Wenn man den Weg zu einer solchen Lösung (in der Komplexitätstheorie auch *Zertifikat* genannt) kennt, dann kann man die Lösung schnell, d.h. in polynomieller Zeit berechnen. Typischerweise sind die Zertifikate **NP**-vollständiger Probleme natürlich. Bei dem SAT-Problem (Erfüllbarkeit aussagenlogischer Formeln) z.B. enthält solch ein Zertifikat die erfüllende Belegung.

Diesen Zusammenhang nutzen wir bei der Untersuchung der Implementierungskorrektheit für die Codeerzeugung aus. Wir modifizieren das bisherige Checker-Szenario so, dass der Compiler neben dem Zielprogramm auch ein Protokoll erzeugt, in dem festgehalten ist, wie die Lösung berechnet wurde. Der Checker erhält dieses Protokoll als dritte Eingabe und verfährt damit wie folgt: Er berechnet aus dem Eingabeprogramm mithilfe des Zertifikats ein Zielprogramm, vergleicht das selbst berechnete Zielprogramm mit dem vom Compiler berechneten und gibt im Fall der Übereinstimmung eine "ja"-Antwort, im negativen Fall dagegen das Ergebnis "weiß nicht". Diese Prüfmethode bezeichnen wir als *Programmprüfen mit Zertifikaten*.

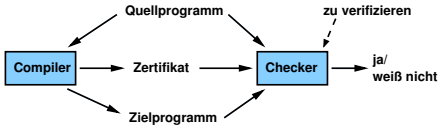


Abbildung 4: Programmprüfen mit Zerifikaten

Man mag sich fragen, was passiert, wenn der Compiler ein fehlerhaftes Zertifikat ausgibt. Wenn der Checker mit solch einem inkorrekten Zertifikat ein korrektes Ergebnis berechnet und wenn weiterhin dieses Ergebnis mit dem vom Compiler berechneten übereinstimmt, dann hat es der Checker geschafft, das vom Compiler berechnete Ergebnis zu verifizieren. Dabei spielt es keine Rolle, wie der Compiler sein Ergebnis ermittelt hat, solange es der Checker mit seiner (verifizierten) Implementierung rekonstruieren konnte.

Betrachten wir noch einmal die Rolle von Codegenerator und Checker genauer. Der Codegenerator verhält sich wie eine nicht-deterministische Turingmaschine, indem er die Lösung sucht und berechnet. Der Checker dagegen agiert wie eine deterministische Turingmaschine und berechnet die Lösung, und zwar auf dieselbe Weise wie der Codegenerator mithilfe des Zertifikats. Das bedeutet, dass man erwarten kann, dass die Implementierung des Checkers mit einem Teil der Implementierung des Codegenerators übereinstimmt, nämlich mit dem Teil, der die Lösung berechnet. Diese Erwartung haben wir in unseren Experimenten bestätigt gefunden. Auf den ersten Blick mag das erstaunlich wirken, weil man ursprünglich Checker verwendet, um einen von der Implementierung unabhängigen Test auf Korrektheit durchführen zu können. Hier hat sich das Prüfparadigma geändert. Wir verwenden den Checker nicht für solch einen unabhängigen Test, sondern als eine Methode, um den korrektheitskritischen Anteil einer Implementierung zu separieren. Die Berechnung der Lösung ist korrektheitskritisch, die Suche nach einer guten Lösung ist es nicht, denn die Güte einer Lösung beeinflusst ihre Korrektheit nicht.

	Codegenerator	Checker
lines of code in .h-Files	949	789
lines of code in .c-Files	20887	10572
total lines of code	21836	11361

Nebenstehende Tabelle zeigt unsere experimentellen Ergebnisse in Zahlen. Wir haben mittels der Methode des Programmprüfens mit Zertifikaten einen Programmprüfer für einen Codegenerator in einem industriellen Projekt entworfen und implementiert [Gl03a, Gl03b]. Der Codegenerator umfasst mehr als 20.000 loc, der Checker nur ungefähr die Hälfte. Bereits daran sieht man, dass wir den Verifikationsaufwand deutlich verringern konnten. Wenn man außerdem bedenkt, dass im Checker die fehleranfällige und aufwändig zu verifizierende Suche nach einer optimalen bzw. guten Lösung *nicht* enthalten ist, erkennt man, dass wir mit der Methode des Programmprüfens mit Zertifikaten den Verifikationsaufwand deutlich verringern konnten.

5 Verwandte Arbeiten

Frühe Arbeiten zur Verifikation von Compilern, insbesondere der Übersetzung der Programmiersprache Piton, wurden im Boyer-Moore-Beweiser durchgeführt [Mo89]. Das von der DFG geförderte *Verifix*-Projekt, das an den Universitäten in Karlsruhe, Kiel und Ulm durchgeführt wurde, hat Methoden entwickelt, mit denen formal korrekte Übersetzer konstruiert werden können, ohne dass dabei Leistungseinbußen entstehen, siehe [GGZ04] für einen Überblick. Einige neuere Arbeiten wie z.B. die Verifikation der Übersetzung von Java in Java Byte Code [KN03] haben sich auf die Übersetzungen im Frontend-Bereich konzentriert. Der Ansatz von beweistragendem Code (*proof-carrying code*) [Ne97] ist schwächer als der von uns verfolgte, weil er sich auf die Verifikation von notwendigen, nicht aber hinreichenden Korrektheitsbedingungen konzentriert. Programmprüfung wurde im Kontext algebraischer Probleme entwickelt [BK95] und wird nicht nur bei der Verifikation von Compilern, sondern auch z.B. bei der Hardware-Verifikation [GD04] eingesetzt.

6 Zusammenfassung und Ausblick

Bei der Verifikation optimierender Compiler sind sowohl logische als auch softwaretechnische Probleme zu lösen. Zum einen muss verifiziert werden, dass die angewandten Transformationsalgorithmen die Semantik der übersetzten Programme erhalten (*Übersetzungskorrektheit*). Zum anderen muss sichergestellt werden, dass diese Algorithmen in einem Compiler auch korrekt implementiert sind (*Implementierungskorrektheit*). Wir haben in unseren Arbeiten gezeigt, wie Übersetzungskorrektheit für optimierende Codeerzeugung ausgehend von SSA-basierten Zwischensprachen, einer modernen Zwischendarstellung in optimierenden Compilern, gezeigt werden kann und haben dazu zwei alternative Beweisansätze vorgestellt und verglichen. Außerdem haben wir die von uns entwickelte Methode des Programmprüfens mit Zertifikaten dargestellt, mit der wir die Korrektheit von Lösungen in Optimierungsproblemen sicherstellen. Mit unseren Ergebnissen haben wir nicht nur dazu beigetragen, optimierende Compiler, die ein wichtiges Werkzeug in der Softwaretechnik sind, zu verifizieren, sondern auch Methoden entwickelt, die allgemein bei der Transformation von Hardware- und Software-Systemen einsetzbar sind. Z.B. wird bei dem Ansatz der *Model Driven Architecture (MDA)* die Systemspezifikation unabhängig von der Systemimplementierung angegeben. Bei der Transformation und Implementierung einer solchen Systemspezifikation werden auch Methoden des Compilerbaus eingesetzt, insbesondere die vorgestellten Verifikationstechniken, mit denen man sicherstellen kann, dass die Implementierung eines Systems korrekt ist bzgl. seiner Spezifikation.

Danksagung: Diese Arbeit wurde von der Deutschen Forschungsgemeinschaft unter dem Geschäftszeichen Gl 360/1-1 sowie von der Landesstiftung Baden-Württemberg im Rahmen des Eliteförderprogramms für Postdoktoranden gefördert.

Literatur

[AZ04] Aigner, M. und Ziegler, G. M.: *Proofs from THE BOOK*. Springer-Verlag. 2004.

[BG04] Blech, J. O. und Glesner, S.: A Formal Correctness Proof for Code Generation from SSA Form in Isabelle/HOL. In: *34. Jahrestagung der GI*. LNI. 2004.

[BGLM04] Blech, J. O., Glesner, S., Leitner, J., und Mülling, S. Some Theorems on Data Dependencies using Partial Orders. 2004. Internal Report, University of Karlsruhe.

[BK95] Blum, M. und Kannan, S.: Designing Programs that Check Their Work. *JACM*. 1995.

[Bo02] Borland/Inprise: *Official Borland/Inprise Delphi-5 Compiler Bug List*. http://www.borland.com/devsupport/delphi/fixes/delphi5/compiler.html. 2002.

[CFR+91] Cytron, Ferrante, Rosen, Wegman, und Zadeck: Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM TOPLAS*. 13(4). 1991.

[CM86] Chirica, L. M. und Martin, D. F.: Toward Compiler Implementation Correctness Proofs. *ACM Transactions on Programming Languages and Systems*. 8(2):185–214. 1986.

[DvHG03] Dold, A., von Henke, F. W., und Goerigk, W.: A Completely Verified Realistic Bootstrap Compiler. *Int'l Journal of Foundations of Computer Science*. 14(4):659–680. 2003.

[GD04] Große, D. und Drechsler, R.: Checkers for SystemC Designs. *Proc. 2nd ACM & IEEE Int'l Conf. on Formal Methods and Models for Codesign (MEMOCODE'2004)*. 2004.

[GFJ04] Glesner, S., Forster, S., und Jäger, M.: A Program Result Checker for the Lexical Analysis of the GNU C Compiler. 2004. Elsevier, Elec. Notes in Theor. Comp. Sc. (ENTCS).

[GGZ98] Goerigk, W., Gaul, T., und Zimmermann, W.: Correct Programs without Proof? On Checker-Based Program Verification. In: *ATOOLS'98*. 1998. Springer.

[GGZ04] Glesner, S., Goos, G., und Zimmermann, W.: Verifix: Konstruktion und Architektur verifizierender Übersetzer. *it - Information Technology*. 46:265–276. 2004.

[Gl03a] Glesner, S.: Program Checking with Certificates: Separating Correctness-Critical Code. In: *12th Int'l FME Symposium (Formal Methods Europe)*. 2003. Springer, LNCS 2805.

[Gl03b] Glesner, S.: Using Program Checking to Ensure the Correctness of Compiler Implementations. *Journal of Universal Comp. Sc. (J.UCS)*. 9(3):191–222. March 2003.

[HGG+99] Heberle, A., Gaul, T., Goerigk, W., Goos, G., und Zimmermann, W.: Construction of Verified Compiler Front-Ends with Program-Checking. In: *Perspectives of System Informatics, Third Int'l A. Ershov Memorial Conf.*. 1999. Springer, LNCS 1755.

[KN03] Klein, G. und Nipkow, T.: Verified Bytecode Verifiers. *TCS*. 298:583–626. 2003.

[Mo89] Moore, J. S.: A Mechanically Verified Language Implementation. *Journal of Automated Reasoning*. 5(4):461–492. 1989.

[Ne97] Necula, G. C.: Proof-Carrying Code. In: *POPL'97*. 1997. ACM.

[Ne01] Newsticker, H.: *Rotstich durch Fehler in Intels C++ Compiler*. http://www.heise.de/newsticker/data/hes-11.11.01-000/. 2001.

[NPW02] Nipkow, T., Paulson, L. C., und Wenzel, M.: *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, Lecture Notes in Computer Science, Vol. 2283. 2002.

[Pa94] Papadimitriou, C. H.: *Computational Complexity*. Addison-Wesley. 1994.

[Po81] Polak, W.: *Compiler Specification and Verification*. Springer, LNCS 124. 1981.

[PSS98] Pnueli, A., Siegel, M., und Singerman, E.: Translation validation. In: *Proc. of Tools and Algorithms for the Construction and Analysis of Systems*. 1998. Springer, LNCS 1384.

[SA97] Schellhorn, G. und Ahrendt, W.: Reasoning about Abstract State Machines: The WAM Case Study. *Journal of Universal Computer Science*. 3(4):377–413. 1997.