Praktikable Konstruktion korrekter Übersetzer

Wolfgang Goerigk[1] Wolf Zimmermann[2] Thilo Gaul[2] Andreas Heberle[2]
Ulrich Hoffmann[1]

[1]Institut für Informatik und Praktische Mathematik
Christian-Albrechts-Universität zu Kiel

[2]Institut für Programmstrukturen und Datenorganisation
Universität Karlsruhe

"Moral: You can't trust code that you did not totally create yourself. (Especially code from companies that employ people like me.) No amount of source-level verification or scrutiny will protect you from using untrusted code."

Ken Thompson
ACM Turing Award Lecture 1984

Zusammenfassung
Wir zeigen, wie man korrekte Übersetzer konstruiert, die auch sinnvolle und gut eingeführte Standard-Übersetzerbautechniken verwenden. Das führt dazu, daß die Qualität des durch den korrekten Übersetzers generierten Codes vergleichbar mit der von unverifizierten Übersetzern generierten Codes wird. Operationale Semantik, ein adäquater Korrektheitsbegriff und Simulationsbeweistechniken lassen ausreichend Freiraum für den Übersetzerkonstrukteur und erlauben eine Modularisierung des Korrektheitsproblems und weitgehende mechanische Beweisunterstützung. Dies kommt einem ingenieurmäßigeren Vorgehen bei Übersetzerkonstruktion und -korrektheitsbeweis entgegen. Insbesondere ist es nicht notwendig, alle komplexen Algorithmen in Übersetzern verifizieren. Stattdessen können ihre Resultate algorithmisch überprüft werden.

1 Einführung

Der Einsatz rechnerbasierter Systeme im Bereich sicherheitskritischer Anwendungen verlangt, daß Software als korrekt nachgewiesen wird. Die Korrektheit der Ausführung hängt aber ganz entscheidend von der Korrektheit des auf der Maschine tatsächlich ausgeführten Codes und damit von dem Übersetzer ab, der diesen Code generiert. Ziel des *Verifix*-Projektes[1] ist es, Methoden zur Konstruktion korrekter Programmiersprachübersetzer für praktisch relevante Sprachen und reale Maschinen zu entwickeln und bis hinab zum binären Maschinencode zu verifizieren, wobei die Qualität des erzeugten Codes mit der von unverifizierten Übersetzern erzeugten Codes vergleichbar ist. Um dieses Ziel zu erreichen fordern wir, daß auch hier Standardübersetzerbautechniken und Architekturen eingesetzt werden. Die Konstruktion eines korrekten Übersetzers erfolgt in drei Schritten:

1. Konstruktion einer korrekten Transformation von Quell- in Zielcode,
2. korrekte Implementierung dieser Transformation in höherer Übersetzerimplementierungssprache und
3. Verifikation des Übersetzerprogramms in binärem Maschinencode der Wirtmaschine

Nur im ersten Schritt ist es notwendig, die Quell- und Zielsprachsemantik zu betrachten. Der zweite Schritt ist eine klassische Programmverifikationsaufgabe. Allerdings sind Übersetzer keine "kleinen" Programme, die man so nebenbei beweisen kann. Wir erreichen die Praktikabilität der Verifikation einer Übersetzerimplementierung bzgl. einer Transformation dadurch, daß viele komplexe Algorithmen, generell und speziell in Übersetzern, gar nicht verifiziert werden müssen, weil ihre Resultate, gleichwohl durch komplizierte Programmteile generiert, häufig leicht algorithmisch überprüfbar sind und durch Überprüfungsroutinen im Code des Übersetzers abgesichert werden können [10, 26]. Die zusätzlich nötige und durchgeführte Verifikation der Übersetzerimplementierung in Maschinencodes schließt das Vorhandensein trojanischer Pferde, vor denen Ken Thompson in [24] warnt, aus.

In der vorliegenden Arbeit diskutieren wir die ersten beiden Schritte. Die Korrektheit der Transformation wird durch Simulationsbeweise, basierend auf operationaler Semantikdefinition mit dynamischen Algebren, bewiesen. Als Beispiel verwenden wir aus Platzgründen die Übersetzung einer einfachen **while**-Sprache in DEC-Alpha-Maschinencode. Die hier vorgestellten Techniken lassen sich jedoch auf realistische Programmiersprachen übertragen. Wir skizzieren deshalb Verallgemeinerungen, wenn diese nicht a priori naheliegend sind. Abschnitt 2 definiert dynamische Algebren, operationale Semantiken und den Korrektheitsbegriff. Abschnitt 3 diskutiert die korrekte Transformation von Quell- und Zielcode und Abschnitt 4 zeigt die Konstruktion einer korrekten Implementierung dieser Transformation. Insbesondere wird dort gezeigt, daß zur Korrektheit eines Übersetzers nur ein kleiner Teil verifiziert werden muß. Abschnitt 5 vergleicht unseren Ansatz mit den wichtigsten anderen Methoden zur Konstruktion korrekter Übersetzer.

[1]Die Arbeit im *Verifix*-Projekt ("Korrekte Übersetzer") wird von der Deutschen Forschungsgemeinschaft (DFG) gefördert.

2 Grundlagen

Die Grundlage zur Definition operationaler Semantiken sind dynamische Algebren. Die Grundidee besteht darin, daß Zustände verschiedenen Interpretationen (Algebren) einer Signatur entsprechen und Zustandsübergangregeln eine Zustandsübergangsrelation definieren. Basierend auf dieser Begriffsbildung wird die Korrektheit von Übersetzern definiert. Wir verwenden hier nur den Ausschnitt aus den dynamischen Algebren, der im Beitrag benötigt wird. Für eine allgemeinere Diskussion verweisen wir auf die Literatur, z.B. [11].

Dynamische Algebren Eine *dynamische Algebra* oder *abstrakte Zustandsmaschine* ist ein Tupel $(\mathcal{S}, \Delta, CT, Z, I)$, wobei $\mathcal{S}$ eine Σ-Algebra mit Trägermenge $\mathbb{X}$ (die *statische Algebra*), Δ die Signatur der *dynamischen Konstanten* , $\Delta \cap \Sigma = \emptyset$, $CT \in \Delta$ ein Befehlszeiger ist, Z eine Menge von *Zustandsübergangsregeln* und I eine Menge von Aktualisierungen (*Initialisierungen*) ist. Ein einstelliges Prädikat über $\mathbb{X}$ heißt *Sorte* und definiert eine Teilmenge von $\mathbb{X}$. Eine *dynamische Sorte* S ist eine mengenwertige dynamische Konstante. Es gibt ein *Superuniversum* $R \in \Sigma$ bestehend aus einer unendlichen Menge von Konstanten. Eine *Aktualisierung* hat die Form $c := t$ für ein $c \in \Delta$ und Term t über der Signatur $\Sigma \cup \Delta$ (bei Initialisierungen sind die rechten Seiten Σ-Terme). *Zustandsübergangsregeln* haben die Form

if $CT \in T \wedge cond$ **then**
  Updates
end

definiert, wobei $T \in \Sigma$ eine Sorte, *cond* ein boolescher Term und *Updates* eine Menge von *Aktualisierungen* ist. Die linken Seiten der Aktualisierungen einer Zustandsübergangsregel sind paarweise verschieden.

Beispiel:[Ausschnitt aus DEC-Alpha-Spezifikation] Die dynamischen Funktionen der DEC-Alpha sind ein Speicher $mem : QUAD \rightarrow BYTE$, Register $reg : BIT^5 \rightarrow QUAD$ und ein Befehlszeiger $PC : QUAD$, wobei $QUAD$ die Menge aller 64-Bit Zahlen ist. Wenn das Programm *prog* ab Adresse a gespeichert ist und die Länge l hat, dann hat die dynamische Algebra die folgenden Initialisierungen:

$$\begin{aligned} mem &:= choose(QUAD \setminus \{a, \ldots, a+l-1\} \rightarrow BYTE) \\ &\quad \cup prog \\ reg &:= choose(\{0, \ldots, 30\} \rightarrow QUAD) \cup \{(31, 0)\} \\ PC &:= a \end{aligned}$$

$choose(X)$ wählt nicht-deterministisch ein Element aus der Menge X. In diesem Beispiel modelliert die Definition mittels *choose* den real anzutreffenden Umstand, daß sowohl Speicher (außer *prog*) als auch Register eine beliebige (indeterministisch gewählte) Vorbelegung haben.

Der Befehl LD $R_i, x(R_j)$ lädt den Inhalt der Speicherzellen $reg(R_j) + x, \ldots, reg(R_j) + x + 7$ in Register R_i. Die entsprechende Zustandsübergangsregel lautet

if $content(PC) \in$ LD **then**
  $reg(dest) := content(rel\_addr + reg(src))$
  $PC := PC + 4$

wobei $content(a)$ die Konkatenation des Inhalts der Speicherzellen $a, \ldots, a + 7$, *dest* den ersten Operanden von $content(PC)$, *rel_addr* die Relativadresse x des Befehls und *src* den zweiten Operanden bezeichnet. ■

Ein *Zustand* q ist eine $\Sigma \cup \Delta$-Algebra über der Trägermenge $\mathbb{X}$, wobei für alle Zustände q die Einschränkung von q auf die Signatur der statischen Funktionen Σ $q|_\Sigma = \mathcal{S}$ ist. $[\![t]\!]_q$ ist die Interpretation eines Terms t über der Signatur $\Sigma \cup \Delta$ im Zustand q. Insbesondere ist $[\![f]\!]_q \in \mathbb{X}$ für alle $f \in \Delta$. Ein Zustand ist *initial*, wenn $[\![f]\!]_q = [\![t]\!]_\mathcal{S}$ für alle Aktualisierungen $f := t \in I$ ist.

Eine Zustandsübergangsregel *feuert* im Zustand q, wenn $[\![CT \in T \wedge cond]\!]_q = true$ ist. Eine feuernde Zustandsübergangsregel definiert zu einem Zustand q einen Nachfolgezustand q' wie folgt: für jede Aktualisierung $c := t$ ist $[\![c]\!]_{q'} = [\![t]\!]_q$. Ein Zustand $q \in Q$ ist *final*, wenn es keine Zustandsübergangsregel gibt, die feuert. Zustandsübergangsregeln definieren eine Zustandsübergangsrelation $\rightarrow$. Eine (endliche oder unendliche) Folge von Zuständen $\langle q_i : 0 \leq i < n \rangle$, $n \in \mathbb{N} \cup \{\infty\}$ heißt *Berechnung*, wenn q_0 initial und $q_i \rightarrow q_{i+1}$ gilt.

Operationale Semantik Die Semantikdefinition basiert wie üblich auf der abstrakten Syntax und besteht aus einem statischen und einem dynamischen Teil. Die statische Semantik wird mittels attributierter Grammatiken beschrieben, die dynamische Semantik durch eine dynamische Algebra.

Eine operationale Semantik einer Programmiersprache ist ein Paar $\mathcal{O} = (AG, \mathcal{A})$, wobei $AG = (G, A, R, B)$ eine attributierte Grammatik, $\mathcal{A} = (\mathcal{S}, \Delta, CT, Z, I)$ eine dynamische Algebra ist und $G = (N, T, P, S)$ eine kontextfreie Grammatik (*abstrakte Syntax*) ist, sowie folgende Eigenschaften erfüllt sind:

(i) Die Algebra der durch AG definierten attributierten Strukturbäume ist eine Unteralgebra von $\mathcal{S}$ und die Symbole $X \in N \cup T \subset \Sigma$ definieren Sorten (Attribute entsprechen Funktionen von Strukturbaumknoten zum Wertebereich der Attribute).

(ii) Für jedes $X \in N$ ist $X.Initial$, $X.Terminal$, $X.NT \in A$.

(iii) Für jedes Programm π ist
  $CT := choose(\pi.Initial) \in I$.

(iv) Alle Zustandsübergangsregeln haben die Form

  if $CT \in X \wedge cond$ **then**
    Updates

  wobei $X \in N \cup T$ ist.

Die Grundidee ist, daß Strukturbaumknoten Befehlen entsprechen können. Das Attribut *Initial* enthält die Menge aller Befehle, über die eine Phrase betreten wird, das Attribut *Terminal* gibt die Menge aller Befehle an, über die eine Phrase verlassen wird. Jeder Befehl hat einen oder mehrere Nachfolgebefehle.

Wenn er nur einen Nachfolgebefehl hat, so ist dieser im Attribut NT enthalten. Jedes Programm $\pi \in L$ einer Programmiersprache L definiert so eine dynamische Algebra, die durch die operationale Semantik von L gegeben ist. Eine dynamische Funktion $f : X \to U$ für $X \in N \cup T$ heißt *dynamisches Attribut* und wird genauso wie statische Attribute notiert.

Bemerkungen:

1. Anstelle attributierter Grammatiken könnte auch jede andere Spezifikationssprache zur Definition statischer Semantik verwendet werden. Für die weitere Vorgehensweise ist nur wichtig, daß nach der semantischen Analyse ein attributierter Strukturbaum vorliegt.
2. Man beachte, daß wir mit dem Begriff *Zustand* sowohl den *Speicherzustand* als auch den *Programmzustand* (Befehlszeiger, CT) zusammenfassen und die Zustandsübergangsregeln auch den Programmzustand modifizieren. ■

Montages Einzelne *Montages* sind den Produktionsregeln der abstrakten Syntax zugeordnet (im Gegensatz zu [14], wo sie konkreter Syntax zugeordnet sind) und bestehen aus vier Komponenten: der Produktionsregel, Attributierungsregeln, einer Bedingung und evtl. Zustandsübergangsregeln (wenn die linke Seite der Produktion ein Befehl ist) zur Spezifikation der dynamische Semantik.

Der Teil der Attributierungsregeln, die die Befehlsstruktur (*Task*-Struktur) definieren, wird durch einen gerichteten Graphen notiert, der sowohl den Steuerfluß als auch den Datenfluß beschreibt. Jede Ecke des Graphen entspricht einem Symbol der abstrakten Syntax. Es gibt *Befehle* (Strukturbaumknoten, die elementaren Befehlen entsprechen) und *Phrasen* (Strukturbaumknoten, die für einen ganzen Teilgraphen stehen), die durch runde bzw. rechteckige Knoten unterschieden werden. Es gibt zwei Arten von Kanten: Steuerflußkanten (gestrichelt) und Datenflußkanten (durchgezogen). Sind Kanten e mit einem Symbol annotiert, so wird damit jeweils der Name eines Attributs des Grammatiksymbols an der Quelle von e definiert. Es kann außerdem zwei ausgezeichnete Ecken I und T geben. Die initialen Befehle sind die Nachfolger von I, die terminalen die Vorgänger von T. Deshalb sind die Ausgangskanten von I bzw. die Eingangskanten von T unmarkiert. Die Graphen in den Montages definieren für jedes Programm einen Steuer- und Datenflußgraphen, dessen Ecken Befehle sind.

$$
\begin{array}{rl}
\mathit{OBJ} & := \emptyset \\
\mathit{env} : \mathit{NAME} \to \mathit{OBJ} & := \emptyset \\
\mathit{content} : \mathit{OBJ} \to \mathit{INT} & := \emptyset \\
\mathit{CT} : \mathit{TASK} & := \mathit{choose}(\pi.\mathit{Initial})
\end{array}
$$

Abbildung 1: Dynamische Konstanten und Initialzustand

Beispiel: Als Beispiel verwenden wir eine einfache **while**-Sprache, deren Programme aus einem Deklarationsteil und einem Anweisungsteil bestehen. Alle Variablen sind vom Typ INT der ganzen Zahlen auf der Zielmaschine. Die Anweisungen sind Schleifen und Zuweisungen. Der Zustand enthält eine Menge $OBJ \subseteq R$ von Objekten (Speicherplätze), eine Umgebung *env*, die Variablennamen Objekte zuordnet, einen Speicher *content*, der wiederum den Objekten ganzzahlige Werte zuordnet, sowie einen Eingabe- und einen Ausgabestrom. Abbildung 1 zeigt den Initialzustand.

Abbildung 2 zeigt die Definition von Variablendeklaration, Schleife und Zuweisung. Ein Deklarationsbefehl erzeugt eine neue Adresse, die aus R genommen wird und bindet diese in der Umgebung an den Bezeichner. Die zwei Attributierungsregeln der Montage für die Schleife sorgen dafür, daß Information aus den Variablendeklarationen in die Ausdrücke und Anweisungen propagiert werden. Die Schleife wird betreten, wenn an dem *loop*-Befehl der Wert der Komponente $CT.cond$ größer 0 ist; ist $CT.cond$ kleiner oder gleich 0, wird die Schleife verlassen. Ist $CT.cond$ undefiniert, was beispielsweise durch arithmetische Überläufe hervorgerufen werden kann, ist kein Zustandsübergang definiert. *Assign*-Tasks enthalten Komponenten *dest* (destination) und *src* (source), die den Speicherplatz der Variablen bzw. den Wert des Ausdrucks enthalten werden, und die Zuweisung modifiziert den Speicher entsprechend. Die Tasks $CT.dest$ enthalten das dynamische Attribut *addr*, das bei Ausführung die Zieladresse der Zuweisung berechnet. Analog enthalten die Tasks $CT.src$ das dynamische Attribut *val* zur Berechnung der Resultate der Ausdrucksauswertung.

Die vollständige Spezifikation der Sprache und deren Übersetzung nach DEC-Alpha befindet sich in einem technischen Bericht (http://i44www.info.uni-karlsruhe.de/~verifix/papers/while.ps) ■

Bemerkung: In **while** wäre die Umgebung *env* nicht notwendig. Bei Sprachen mit Blockschachtelung und rekursiven Prozeduren ist sie jedoch notwendig. Die Zustandsübergänge der Deklaration sowie der Zuweisung bleiben jedoch dieselben. ■

Korrektheitsbegriff Ein Übersetzer überführt Programme einer Quellsprache L_1 in Programme einer Zielsprache L_2. Wir setzen den Quellzustandsraum Q_1 und den Zielzustandsraum Q_2 durch eine Repräsentationsrelation $\rho \subseteq Q_1 \times Q_2$ in Beziehung. Ein Zielprogramm $\pi_2 \in L_2$ heißt *korrekte Implementierung von* $\pi_1 \in L_1$, falls für alle initialen Zustände $i_1 \in Q_1, i_2 \in Q_2$ und finale Zustände $f_2 \in Q_2$ gilt:

$$
\begin{array}{l}
(i_1, i_2) \in \rho \text{ und } i_2 \to^*_{\pi_2} f_2 \Longrightarrow \\
\text{es gibt ein finales } f_1 \in Q_1 \text{ mit } (f_1, f_2) \in \rho \\
\text{und } i_1 \to^*_{\pi_1} f_1
\end{array}
$$

Das Zielprogramm, angesetzt auf die Repräsentation i_2 eines Initialzustandes i_1, endet also dann in einem regulären Finalzustand f_2, wenn das Quellprogramm, angesetzt auf i_1, in f_1 endet und f_2 Repräsentation von f_1 ist (*Erhalten partieller Korrektheit*). Dabei ist ein Finalzustand $f \in Q_2$ *regulär*, wenn es ein $q \in Q_1$ gibt, so daß $(q, f) \in \rho$ ist. Wenn beispielsweise das Zielprogramm mit einer Division durch 0 terminiert, so ist der Finalzustand regulär, weil auch ein Programm in der Quellsprache mit Division durch 0

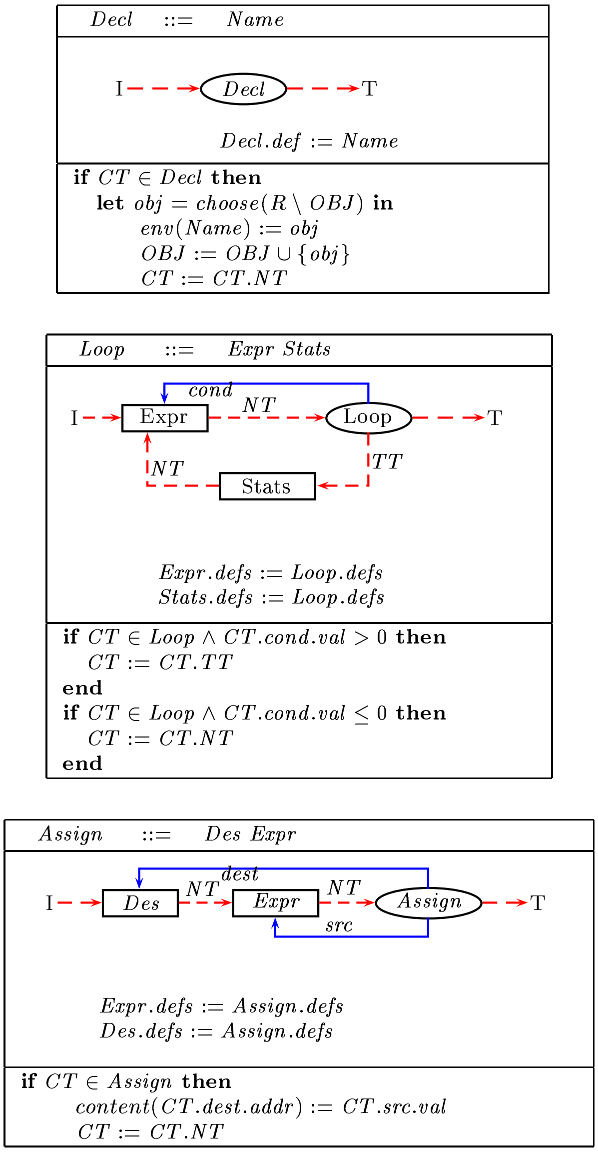

Abbildung 2: Montages für die **while**-Schleife und Zuweisung

terminieren kann. Umgekehrt kann ein Zielprogramm terminieren, wenn der Speicher überläuft. Der entsprechende Zustand ist irregulär, wenn die Quellsprache keine Speicherplatzbeschränkungen kennt.

3 Übersetzung

Eine korrekte Transformation der Quell- in die Zielsprache kann nach folgender Methode konstruiert werden (vgl. Abbildung 3): zuerst wird für die Quellsprache durch die Speicherabbildung eine andere Semantik im Speicherzustandsraum der Zielsprache angegeben. Anschließend wird innerhalb eines Zustandsraum das Programm transformiert. Schließlich wird der Befehlszeiger in den Zustandsraum der Zielsprache abgebildet. Führt man zur Übersetzung einer Quellsprache L in eine Zielsprache L' Zwischensprachen L_i, $i = 1, \ldots, n$ ein, so wählt man diese Vorgehensweise für jede Übersetzung von L_i nach L_{i+1}, $i = 0, \ldots, n$ ($L_0 = L$ und $L_{n+1} = L'$). Wir gehen davon aus, daß das Programm bereits als attributierter Strukturbaum vorliegt, und demonstrieren die Vorgehensweise anhand der Übersetzung der Deklaration, der Schleife und der Zuweisung. Der Vorteil

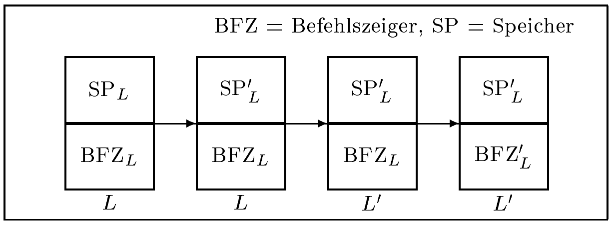

Abbildung 3: Übersetzung einer Sprache L nach L'

unserer Vorgehensweise liegt darin, daß die Korrektheit von Transformationen innerhalb des gleichen Zustandsraumes durchgeführt werden können. Deshalb gelten die Ausführungen in Unterabschnitt 3.2 genauso für optimierende Transformationen.

3.1 Speicherabbildung

Neben der Quellsprachsemantik $\mathcal{O} = (AG, \mathcal{A})$ wird durch eine Speicherabbildung für die Quellsprache eine Semantik $\mathcal{O}_z = (AG, \mathcal{B})$ im Speicherzustandsraum der Zielmaschine festgelegt. Die Speicherabbildung induziert für jedes Programm π eine partielle injektive Abbildung $\phi : Q_1 \to Q_2$, wobei Q_1 der Zustandsraum von $\mathcal{A}$ und Q_2 der Zustandsraum von $\mathcal{B}$ ist. Ziel ist es, die Speicherabbildung so zu definieren, daß für alle Zustände $q, q' \in RAN(\phi)$ das folgende Diagramm kommutiert:

$$\begin{tikzcd}
\phi^{-1}(q) \arrow[r, "1"'] & \phi^{-1}(q') \\
q \arrow[u, "\phi^{-1}"] \arrow[r, "2"'] & q' \arrow[u, "\phi^{-1}"]
\end{tikzcd}$$

Hier ist $\to_2$ die durch $\mathcal{B}$ und $\to_1$ die durch $\mathcal{A}$ definierte Zustandsübergangsrelation. Man beachte, daß an dieser Stelle *keine* Programmtransformation stattfindet. Die Semantik $\mathcal{O}_z$ der Sprache L entspricht bis auf Ressourcenbeschränkung 1-1 der originalen Semantik $\mathcal{O}$ von L. Damit ist auch die Korrektheitsdefinition aus Abschnitt 2 erfüllt.

Im Falle der Übersetzung von **while**-Programmen nach DEC-Alpha[23] Maschinencode ist die Speicherabbildung eine Abbildung $reladdr : NAME \to ADDR \cup REG$, wobei $NAME$ die Menge aller Namen im Deklarationsteil des zu übersetzenden Programms, $ADDR$ die Menge der Relativadressen der DEC-Alpha Prozessorfamilie und REG die Menge der Register ist. Der Speicher mem ist byte-orientiert, der Adressbereich $ADDR$ sind 64-Bit vorzeichenbehaftete ganze Zahlen, der Registerbereich REG sind 5-Bit natürliche Zahlen. Da arithmetische Operationen nur auf Registern ausgeführt werden können, benötigt man noch einen Akkumulator, den wir in Register R_1 verwalten, R_0 enthält in unserem Übersetzungsbeispiel immer die Datenbasisadresse[2].

[2] Die Wahl der konkreten Register ist natürlich beliebig

Annahme 1 *reladdr ist injektiv. Falls $reladdr(x) \in ADDR$ ist, so hat diese die Ausrichtung 4. Falls $reladdr(x) \in REG$ ist, so ist $reladdr(x) \notin \{0, 1, 31\}$.*

Die Zustandsübergangsregeln verwenden nun anstelle der Sorte *OBJ* Adressen und Register. Die Umgebung *env* wird durch *reladdr* ersetzt und wird damit statisch. Anstelle der dynamischen Funktion *content* werden nun der Speicher und die Register des DEC-Alpha-Prozessors benutzt. Formal kann man diese neue Semantik aus der alten ausrechnen, so daß die neue Semantik korrekt ist, sofern Annahme 1 erfüllt ist. Abbildung 4 zeigt die Semantik der Schleife und der Zuweisung nach der Speicherabbildung und die Definition für *content* und *env*. *concat* konkateniert 8 Bytes zu einer (vorzeichenbehafteten) ganzen Zahl. $take(n, x)$ berechnet das n-te Byte einer 64-Bit vorzeichenbehafteten ganzen Zahl. Durch einfaches Nachrechnen der Zustandsübergänge kann man zeigen, daß unter Annahme 1 die Speicherabbildung korrekt ist.

Bemerkungen: Die Vorgehensweise ist verallgemeinerbar. Hat man beispielsweise rekursive Prozeduren und Funktionen, so hat *reladdr* zwei Argumente: der Funktionsname und die Parameter und lokalen Variablen der Funktion. Die Umgebung wird in den Speicher abgebildet. Dadurch ergeben sich weitere Anforderungen. Diese lassen sich entweder direkt beweisen oder wie z.B. Annahme 1 während der Übersetzung überprüfen. [26] stellt ausführlich die Konstruktion korrekter Back-Ends mittels Termersetzungssystemen und dynamischer Algebren vor. ■

if $CT \in Decl$ **then**
$\quad CT := CT.NT;$

if $CT \in Loop \wedge reg(1) > 0$ **then**
$\quad CT := CT.TT$

if $CT \in Loop \wedge reg(1) \leq 0$ **then**
$\quad CT := CT.NT$

if $CT \in Assign \wedge CT.dest.addr \in REG$ **then**
$\quad reg(CT.dest.addr) := reg(1)$
$\quad CT := CT.NT$

if $CT \in Assign \wedge CT.dest.addr \notin REG$ **then**
$\quad content(reg(0) + CT.dest.addr) := reg(1)$
$\quad CT := CT.NT$

$$env(x) \triangleq \begin{cases} reladdr(x) & \text{falls } reladdr(x) \in REG \\ reg(0) + reladdr(x) & \text{falls } reladdr(x) \in ADDR \end{cases}$$

$$content(a) \triangleq concat(mem(a), \ldots, mem(a+7))$$

$$content(a) := x \triangleq$$
$$\begin{aligned} mem := \quad mem \quad & \setminus \{(a, mem(a)), \ldots, (a+7, mem(a+7))\} \\ & \cup \{(a, take(7, x)), \ldots, (a+7, take(0, x))\} \end{aligned}$$

Abbildung 4: Semantik von **while** nach Speicherabbildung

3.2 Transformation

Der Ausgangspunkt der Transformation ist der attributierte Strukturbaum, insbesondere der Steuer- und Datenflußgraph. Deshalb gibt die Transformation an, wie der Steuer- und Datenflußgraph transformiert wird. Wir betrachten hier der Einfachheit halber speziell eine Transformation $\mathcal{T}_{\text{task}}$, die angibt, wie die Befehle der Quellsprache abgebildet werden. Abbildung 5 gibt die Transformation für die Schleife und die Zuweisung an. In unserem Beispiel entsteht ein Steuerflußgraph mit DEC-Alphabefehlen, d.h. ein Grundblockgraph.

Da bereits eine Semantik im Speicherzustandsraum der Zielmaschine vorliegt, genügt es die Zustandsrelation auf Basis des Befehlszeigers CT zu definieren: $(q, \bar{q}) \in \rho$ genau dann, wenn $[\![CT]\!]_{\bar{q}} = \mathcal{T}^i_{\text{task}}[\![[\![CT]\!]_q]\!]$ und $[\![f]\!]_{\bar{q}} = [\![f]\!]_q$ für alle $f \in \Delta \setminus \{CT\}$, wobei $\mathcal{T}^i_{\text{task}}$ der erste und $\mathcal{T}^t_{\text{task}}$ der letzte Befehl von $\mathcal{T}_{\text{task}}$ ist. Die Korrektheit der Übersetzung folgt dann direkt durch Induktion über die Anzahl der Zustandsübergänge aus dem

Lemma 1 *Für alle $t \in TASK$ und alle Zustände $q, \bar{q}, \bar{q}'$ mit $(q, \bar{q}) \in \rho$ und $\bar{q} \xrightarrow{\mathcal{T}_{\text{task}}[\![t]\!]^*} \bar{q}'$ gibt es einen Zustand q', so daß $q \to q'$ und $(q', \bar{q}') \in \rho$.*

Bemerkungen:

1. Bei Optimierungen hat man allgemeine Graphtransformationen. Lemma 1 kann leicht auf allgemeine Graphtransformationen verallgemeinert werden.
2. In der Transformationsphase verwendet man oft Baumtransformationen. In unserem Kontext bedeutet dies eine Zuordnung von Graphtransformationen zu Produktionen. Das hat den Vorteil, daß man mehr Techniken zum Beweis der Korrektheit einer Transformation einsetzen kann. Einerseits kann man immer noch eine strukturelle Induktion über den Aufbau der abstrakten Syntaxbäume durchführen, andererseits kann man auch die Korrektheit einer Transformation durch Induktion über die Anzahl der Zustandsübergänge beweisen. Es sind sogar Kombinationen dieser beiden Beweistechniken möglich. ■

$$\mathcal{T}_{\text{task}}[\![Loop]\!] = \mathsf{BGT}\ R_1$$

$$\mathcal{T}_{\text{task}}[\![Assign]\!] = \begin{cases} \mathsf{ST}\ R1\ x(R0) \\ \quad \text{falls } x = reladdr(Des.Name) \wedge \\ \quad x \in ADDR \\ \mathsf{ADD}\ R1\ R31\ Rx \\ \quad \text{falls } x = reladdr(Des.Name) \wedge \\ \quad x \in REG \end{cases}$$

Abbildung 5: Transformation nach DEC-Alpha: Anweisungen

3.3 Linearisierung des Programms

Die aus Zielmaschinenbefehlen bestehende Befehlsstruktur muß nun im Speicher abgelegt werden. Dazu ist es notwendig diese Befehlsstruktur zu linearisieren. Jeder Befehl wird auf eine Addresse abgebildet. Die Linearisierung erfolgt in zwei Phasen: ei-

Situation ($mapbef(b) = (a, s)$)	geforderte Eigenschaften
$b \neq \mathsf{BGT} \wedge mapbef(b.NT) = (a+4, s'))$	$s = \langle b \rangle$
$b \neq \mathsf{BGT} \wedge mapbef(b.NT) \neq (a+4, s')$	$s = \langle b; \mathsf{JMP}\ x \rangle \ \wedge mapbef(b.NT) = (a+x, s')$
$b = \mathsf{BGT} \ \wedge mapbef(b.TT) \neq (a+4, s')$ $\wedge mapbef(b.NT) \neq (a+4, s')$	$s = \langle \mathsf{BGT}\ R1\ x_1; \mathsf{JMP}\ x_2 \rangle \ \wedge$ $mapbef(b.TT) = (a+x_1, s_1) \wedge$ $mapbef(b.NT) = (a+4+x_2, s_2)$
$b = \mathsf{BGT} \ \wedge mapbef(b.NT) = (a+4, s')$	$s = \langle \mathsf{BGT}\ R1\ x \rangle \ \wedge mapbef(b.TT) = (a+x, s_1)$
$b = \mathsf{BGT} \ \wedge mapbef(b.TT) = (a+4, s')$	$s = \langle \mathsf{BLE}\ R1\ x \rangle \wedge mapbef(b.NT) = (a+x, s_1)$

Tabelle 1: Linearisierung: Einfügen von Sprungbefehlen

ner Planungsphase, die für jeden Befehl vorberechnet, wo er in den Speicher abgebildet wird (Abbildung $mapbef : TASK \to ADDR \times INSTR^*$), und einer Transformationsphase, die diese Befehle tatsächlich in den Speicher abbildet. Die Grundblöcke müssen disjunkt abgebildet werden, wobei die Befehle in einem Grundblock nacheinander in ihrer Reihenfolge abgebildet werden. Eventuell müssen anschließend an Befehle noch Sprünge eingefügt werden. Wenn beispielsweise in einer Befehlsstruktur ein Befehl b_2 mehr als einen Vorgänger hat, so kann b_2 höchstens unmittelbar hinter einem seiner Vorgänger abgebildet werden; bei allen anderen Vorgängern wird anschließend ein unbedingter Sprung auf b_2 benötigt. Insgesamt darf kein Befehl auf eine Adresse abgebildet werden, die Programmdaten enthalten kann.

Beispiel: Die DEC-Alpha-Maschinensprache erlaubt es, direkt Relativadressen in den Sprungbefehl zu kodieren. Z.B. bewirkt die Ausführung des Befehls $\mathsf{JMP}\ x$ die Aktualisierung $PC := PC + x$. Wir nehmen der Einfachheit halber an, daß nur solche Sprungbefehle verwendet werden. Alle Nichtsprungbefehle erhöhen den Befehlszähler um die Befehlslänge, d.h. $PC := PC + 4$. Erfüllt *mapbef* Annahme 2 so ist die Linearisierung des Grundblockgraphen korrekt.

Annahme 2 *mapbef erfüllt folgende Eigenschaften:*

(i) *Für alle Befehle* b, b' *des DEC-Alpha-Grundblockgraphen mit* $mapbef(b) = (a, s)$ *und* $mapbef(b') = (a', s')$ *gilt:*
$\{a, \ldots, a + (|s| - 1) \cdot 4\} \cap$
$\{a', \ldots, a' + (|s'| - 1) \cdot 4\} \neq \emptyset \ \Rightarrow b = b'$

(ii) *Für alle Befehle* b *des DEC-Alpha-Grundblockgraphen mit* $mapbef(b) = (a, s)$ *und alle* $x \in DOM(reladdr)$ *so daß* $reladdr(x) \in ADDR$ *und* $addr = reg(R_0) + reladdr(x)$ *ist, gilt:*
$\{addr, \ldots, addr + 7\} \cap \{a, \ldots, a + (|s| - 1) \cdot 4\} = \emptyset$

(iii) *mapbef erfüllt die Eigenschaften in Tabelle 1.* ■

4 Implementierung

Die Vorgehensweise in Abschnitt 3 erlaubt die Implementierung eines Übersetzers durch eine klassische Übersetzerarchitektur. Eine komplette Programmverifikation aller Teile des Übersetzers ist im allgemeinen nicht praktikabel. Wir zeigen, daß gar nicht alle Komponenten verifiziert werden müssen, sondern man oft nur das Resultat, das von Komponenten berechnet wird, überprüfen muß. Für eine korrekte Implementierung ist in diesen Fällen nur eine verifizierte Überprüfungsroutine notwendig.

In der ingenieursmäßigen Übersetzerkonstruktion werden hier fast immer optimierende Algorithmen von Generatoren erzeugt oder Bibliotheken entnommen. Besonders wichtig hierbei ist auch, daß die oft auf Heuristiken basierenden Algorithmen ausgetauscht oder parametrisiert werden können, um für eine gegebene Übersetzung dem möglichen Optimum (Laufzeit- oder Speicherverbrauch) näher zu kommen. Vielen dieser Algorithmen ist gemein, einerseits sehr komplex, andererseits aber auch leicht anhand ihres Ergebnisses zu überprüfen zu sein. Typische Vertreter sind Registerallokation, Speicheranordnung und Codelinearisierung, aber auch die eigentlichen Transformationsaufgaben wie Graph- und Baumersetzung.

Annahmen 1 (Abschnitt 3.1) und 2 (Abschnitt 3.3) sind Beispiele für solche Algorithmen. Die Eigenschaften in Annahmen 1 und 2 definieren zu überprüfende Anforderungen an Algorithmen, die die Spezifikationen implementieren und legen gleichzeitig Schnittstellen fest. Die Implementierung dieser Spezifikation stellt eine klassische Optimierungsaufgabe dar. Annahme 2 definiert Anforderungen an die Funktion *mapbef*, die die Abbildung der Task-Graphen in den Speicher vornimmt. Die Implementierung dieser Funktion kann diese Abbildung beliebig nach Längen der Sprungbefehle und Steuerflußwahrscheinlichkeiten optimieren, das Resultat dieser Funktion kann sehr einfach mittels der spezifizierten Anforderungen auf Korrektheit überprüft werden. Gleiches gilt für Annahme 1 und ihre Verallgemeinerungen, eine platzoptimierende Speicherabbildung kann nun für komplexe Datentypen den Speicherverschnitt und/oder Datenzugriffszeiten optimieren. Implementierungen zu einer Spezifikation mit der eben beschriebenen Eigenschaft nennen wir **algorithmisch überprüfbar**. Ebenfalls ist die Transformation in Abbildung 5 algorithmisch überprüfbar, wenn sowohl Quell- wie Zielprogramm festliegen. Ein Generator erzeugt die (nicht-induktive) Transformation, eine Überprüfungsroutine testet anschließend induktiv die Gleichungen.

Um zu einer korrekten Implementierung einer Spezifikation S durch Algorithmus A zu gelangen, reicht es also aus

- eine geeignete Schnittstelle T zu definieren, über die Algorithmus A seine Ergebnisse abliefert

- Algorithmus A in Implementierungssprache P **unverifiziert** zu implementieren

- einen Überprüfungsalgorithmus A' verifiziert zu implementieren, der die Ergebnisse von A an Schnittstelle T gemäß S überprüft.

Bei der Schnittstelle T handelt es sich typischerweise um einfache Annotationen der übergebenen Strukturen, z.B. um Registernummern bei der Registerallokation oder Instruktionssequenzen wie in *Annahme 2*. Leicht einzusehen ist auch, daß Überprüfungsalgorithmen A' wesentlich einfacher zu implementieren und verifizieren sind, als die eigentlichen Optimierungsalgorithmen A. Anforderungen an eine Implementierung dieser sind lediglich zugesicherte Terminierungseigenschaften, sowie Abgeschlossenheit in Steuerfluß und Datenzugriffen bzgl. anderer Programmteile.

Optimierungsalgorithmen können nun aus einer Bibliothek kommen oder sogar von Generatoren aus Spezifikationen erzeugt werden. Diese Generatoren müssen selbst auch nicht verifiziert implementiert sein, da es unerheblich ist, wie das Generierungsergebnis zustande gekommen ist. Es muß nur dafür gesorgt werden, daß die erzeugten Algorithmen korrekt mit den Überprüfungsalgorithmen zusammengesetzt werden.

Eingesetzt werden kann die Technik der algorithmischen Überprüfung also überall dort, wo die Überprüfung des Ergebnisses eines Algorithmus einfacher ist als die Verifikation des Algorithmus selber. Das gilt im besonderen dann, wenn zur Laufzeit aus mehreren Algorithmen und Implementierungen gewählt werden soll oder diese sogar generiert wurden.

5 Verwandte Arbeiten

In anderen Arbeiten zur Konstruktion korrekter Übersetzer werden Ressourcenbeschränkungen in der Korrektheitsdefinition nicht berücksichtigt. Beispielsweise diskutiert [13] eine Reihe aus der Literatur bekannte Korrektheitsdefinitionen von Übersetzern, die aber alle keine Ressourcenbeschränkungen behandeln. Durch Übersetzer generierte Maschinenprogramme können jedoch durch die endlichen Ressourcenbeschränkungen der Hardware auch in Fällen scheitern, in denen das Quellprogramm eine wohldefinierte Bedeutung hat. Wenn reale Zielmaschinen ins Spiel kommen, sind deshalb solche Korrektheitsdefinitionen wie z.B. Bisimulation eine zu starke Forderung.

[15] ist die erste uns bekannte Arbeit zur Konstruktion korrekter Übersetzer. Die meisten folgenden Arbeiten über korrekte Übersetzer basieren auf denotationalen Semantiken (z.B. [4, 16, 17, 20, 21, 25]), schrittweiser Verfeinerung (z.B. [5, 6, 12, 18, 3, 2, 1]), strukturelle operationale Semantiken (z.B. [7]) und algebraischen Ansätzen (vgl. [22]). Unser Ansatz unterscheidet sich in mehrerer Hinsicht von diesen Ansätzen. Erstens übersetzen wir bis zum Maschinencode, wobei wir nicht auf bestimmte Codeerzeugungstechniken wie Makroexpansion eingeschränkt sind, zweitens erlauben unsere Ansätze neben der Beweistechnik der strukturellen Induktion über den Aufbau von Strukturbäumen auch eine Induktion über die Anzahl der Zustandsübergänge, drittens betrachten wir die Korrektheit der Übersetzerimplementierung, und viertens können wir neben der Korrektheit von Baumtransformationen auch die Korrektheit von allgemeinen Graphtransformationen nachweisen.

Die Beweistechnik der strukturellen Induktion über den Baumaufbau sowie die Korrektheit von allgemeinen Graphtransformationen läßt sich kaum mit baumbasierten Semantiken wie denotationalen oder strukturell operationalen Semantiken bzw. den algebraischen Ansätzen nachweisen. Aus diesem Grund betrachten alle uns bekannten Arbeiten auch nur die Übersetzung in eine immer noch sehr hohe Zwischensprache die dann interpretiert wird. Die Effizienz des erzeugten Codes ist um mehrere Größenordnungen schlechter als die von unverifizierten Übersetzern (vgl. [20, 7]). Verfeinerungstechniken transformieren üblicherweise bis zur Maschinensprache, berücksichtigen jedoch nach unserer Erkenntnis keine allgemeinen Graphtransformationen und gehen deswegen nicht wesentlich über Makroexpansion hinaus. Bei den auf dynamischer Semantik basierenden Arbeiten [3, 2, 1] könnten allgemeine Graphtransformationen prinzipiell bewiesen werden, während dies bei den ProCoS-Arbeiten unklar ist [5, 6, 12, 18]. ProCoS betrachtet dafür zumindest teilweise auch die Implementierungskorrektheit.

6 Schlußbemerkungen und Ausblick

Die von uns vorgeschlagenen Methoden versetzen den Softwareingenieur in die Lage, relativ einfach zu einem verifiziert implementierten Übersetzer zu gelangen, der mit Standardtechniken des Übersetzerbaus erstellt wurde und auch komplexe, optimierende Algorithmen nutzt. Der adäquate Korrektheitsbegriff, die einfach zu handhabende operationelle Semantik und die vorgestellten Beweistechniken ermöglichen eine weitgehende Modularisierung der Verifikationsaufgabe, sowie eine mechanische Beweisunterstützung.

Wir haben gezeigt, wie man auch große Programme wie einen Übersetzer korrekt implementieren kann, indem man die Technik der algorithmischen Überprüfung der Ergebnisse einzelner Komponenten einsetzt. Z.B. können Übersetzergeneratoren große Teile eines Übersetzers automatisch erzeugen ohne sie verifizieren zu müssen, wenn die Resultate der generierten Algorithmen durch verifizierte Prozeduren überprüft werden. Zusammengenommen ermöglicht dies, ingenieursmäßige Übersetzerkonstruktion zu betreiben und korrekte Übersetzer zu bauen, deren erzeugter Code in der Laufzeiteffizienz mit der unverifizierter, industriell genutzter vergleichbar ist. Sicherlich läßt sich die Idee der algorithmischen Überprüfung zur Konstruktion großer korrekter Programme auch auf andere Gebiete der Softwaretechnik übertragen.

Laufende Arbeiten beschäftigen sich damit, ein komplettes Rahmenwerk aus Generatoren und Bibliotheken verifizierter Algorithmen zu erstellen. In dieses Rahmenwerk wird auch eine Beweisunterstützung mit PVS [19] integriert, die vor allem die einander sehr ähnlichen Beweise lokaler Transformationsregeln mechanisiert [8]. Der Bibliothekskonstruktion kommen Erkenntnisse aus dem Bereich der objektorientierten Softwarekonstruktion zugute [9], hier ist zu erwarten, daß man relativ einfach auf den großen Fundus an Algorithmenbibliotheken zurückgreifen kann.

Literatur

[1] Börger, E. und I. Durdanovic: *Correctness of Compiling Occam to Transputer.* The Computer Journal, 39(1):52–92, 1996.

[2] Börger, E., I. Durdanovic und D. Rosenzweig: *Occam: Specification and Compiler Correctness.Part I: The Primary Model.* In: Montanari, U. und E.-R. Olderog (Herausgeber): *Proc. Procomet'94 (IFIP TC2 Working Conference on Programming Concepts, Methods and Calculi).* North-Holland, 1994.

[3] Börger, E. und D. Rosenzweig: *The WAM-definition and Compiler Correctness.* North-Holland Series in Computer Science and Artificial Intelligence. Beierle, L.C. and Pluemer, L., 1994.

[4] Brown, D. F., H. Moura und D. A. Watt: *Actress: an Action Semantics Directed Compiler Generator.* In: *Compiler Compilers 92*, Band 641 der Reihe *Lecture Notes in Computer Science*, 1992.

[5] Buth, B., K.-H. Buth, M. Fränzle, B. v. Karger, Y. Lakhneche, H. Langmaack und M. Müller-Olm: *Provably correct compiler development and implementation.* In: Kastens, U. und P. Pfahler (Herausgeber): *Compiler Construction*, Band 641 der Reihe *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[6] Buth, B. und M. Müller-Olm: *Provably Correct Compiler Implementation.* In: *Tutorial Material – Formal Methods Europe '93*, Seiten 451–465, Denmark, April 1993. IFAD Odense Teknikum.

[7] Diehl, S.: *Semantics-Directed Generation of Compilers and Abstract Machines.* Doktorarbeit, Universität Saarbrücken, 1996.

[8] Dold, A., T. Gaul und W. Zimmermann: *Mechanized Verification of Compiler Back-Ends.* In: Steffen, B. und T. Margaria (Herausgeber): *Proceedings of the International Workshop on Software Tools for Technology Transfer STTT '98*, Seiten 13–24, 1998.

[9] Frick, A., W. Zimmer und W. Zimmermann: *Konstruktion robuster und flexibler Klassenbibliothekenx.* Informatik - Forschung und Entwicklung, 11(4):168–178, 1996. Eine frühere Version dieses Arikels erschien in *Softwaretechnik'95*, S. 35–46.

[10] Goerigk, W., T. Gaul und W. Zimmermann: *Correct Programs without Proof? On Checker-Based Program Verification.* Accepted for ATOOLS'98 Workshop on "Tool Support for System Specification, Development, and Verification", Malente, 1998.

[11] Gurevich, Y.: *Evolving Algebras: Lipari Guide.* In: Börger, E. (Herausgeber): *Specification and Validation Methods.* Oxford University Press, 1995.

[12] Hoare, C.A.R., He Jifeng und A. Sampaio: *Normal Form Approach to Compiler Design.* Acta Informatica, 30:701–739, 1993.

[13] Janssen, T.M.V.: *Algebraic Translations, Correctness and Algebraic Compiler Construction.* Theoretical Computer Science, 199:25–56, 1998.

[14] Kutter, P. W. und A. Pierantonio: *Montages Specifications of Realisitic Programming Languages.* Journal of Universal Computer Science, 3(5):416–442, 1997.

[15] McCarthy, J. und J.A. Painter: *Correctness of a Compiler for Arithmetical Expressions.* In: Schwartz, J.T. (Herausgeber): *Proceedings of a Symposium in Applied Mathematics, 19, Mathematical Aspects of Computer Science.* American Mathematical Society, 1967.

[16] Mosses, P. D.: *Abstract semantic algebras.* In: Bjørner, D. (Herausgeber): *Formal description of programming concepts II*, Seiten 63–88. IFIP IC-2 Working Conference, North Holland, 1982.

[17] Mosses, P. D.: *Action Semantics.* Cambridge University Press, 1992.

[18] Müller-Olm, Markus: *Modular Compiler Verification*, Band 1283 der Reihe *Lecture Notes in Computer Science.* Springer-Verlag, 1996.

[19] Owre, S., J. M. Rushby und N. Shankar: *PVS: A Prototype Verification System.* In: Kapur, Deepak (Herausgeber): *Proceedings 11th International Conference on Automated Deduction CADE*, Band 607 der Reihe *Lecture Notes in Artificial Intelligence*, Seiten 748–752, Saratoga, NY, Oktober 1992. Springer-Verlag.

[20] Palsberg, J.: *An automatically generated and provably correct compiler for a subset of ADA.* In: *IEEE International Conference on Computer Languages*, 1992.

[21] Paulson, L.: *A compiler generator for semantic grammars.* Doktorarbeit, Stanford University, 1981.

[22] Rus, T.: *Algebraic Processing of Programming Languages.* Theoretical Computer Science, 199:105–143, 1998.

[23] Sites, Richard L.: *Alpha Architecture Reference Manual.* Digital Equipment Corporation, 1992.

[24] Thompson, Ken: *Reflections on Trusting Trust.* Communications of the ACM, 27(8):761–763, 1984. ACM Turing Award Lectures: The First Twenty Years 1965-1985, ACM Press, 1987, and in Computers Under Attack: Intruders, Worms, and Viruses Copyright, ACM Press 1990.

[25] Wand, M.: *A semantic prototyping system.* SIGPLAN Notices, 19(6):213–221, Juni 1984. SIGPLAN 84 Symp. On Compiler Construction.

[26] Zimmermann, W. und T. Gaul: *On the Construction of Correct Compiler Back-Ends: An ASM Approach.* Journal of Universal Computer Science, 3(5):504–567, 1997.