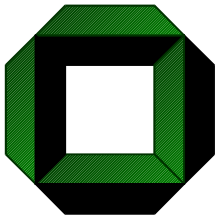
Ausgewählte Kapitel aus dem Übersetzerbau



Prof. G. Goos

Sebastian Hack, Rubino Geiß

Universität Karlsruhe

Adenauerring 20a, Raum 201, 233, 236

Tel. 608-4760, -7399, -8352

{ggoos,hack,rubino}@ipd.info.uni-karlsruhe.de

www.info.uni-karlsruhe.de

Voraussetzung / Lernziele

- Voraussetzungen: Vorlesung Übersetzerbau
  - Zwischendarstellungen: SSA
  - Registerzuteilung
  - Grundkenntnisse in Programmiersprachen

- Lernziele:
  - Übersetzerverifikation
  - Optimierung & Codeerzeugung bei Darstellungen in SSA-Form
  - Methoden der klassischen Codeoptimierung
  - Optimierung für Parallele Programme

Vorlesungsplan

Einführung (1)

Verifikation (1)

Datenflussanalyse (2-5)

- Einführung
- Analysen und Fragestellungen
- Algorithmen

Zwischensprachen mit SSA-Eigenschaft (5)

- Speicherdarstellung
- Chi-Terme
- Lowering / Codegenerierung
- Registerzuteilung
- Nachoptimierung

Cache-Optimierungen (10)

- Fortschrittliche Methoden der Registerzuteilung
- Befehlsanordnung

Parallele Sprachen, Parallelisierung, Threads (12)

Inhalt – Einführung

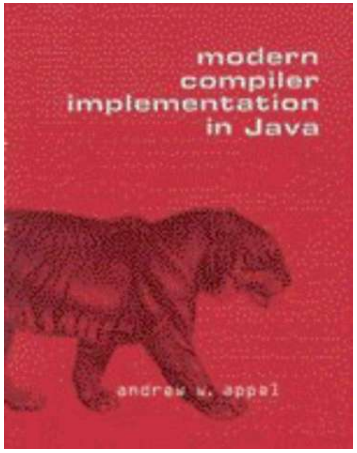
Einführung

- Motivation
- Vorlesungsplan
- Literatur
- Internetseiten – Software

- Bitte um aktive Teilnahme
  Ohne Anregung / Kritik kann die Vorlesung nicht „rund" werden!
  Also bitte Kontakt zu den Herrn Goos, Hack oder Geiss nicht scheuen;
  wie bekannt gibt es nur blöde Antworten, aber keine
  blöden Fragen :)

DANKE

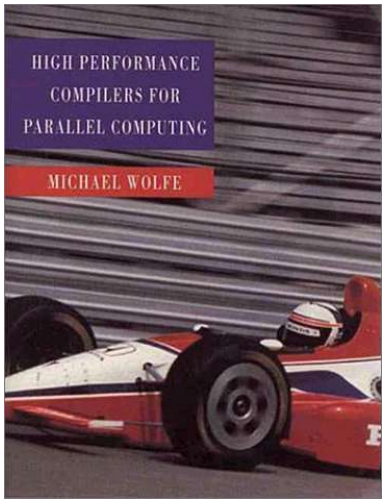
Literatur – Hauptwerke I



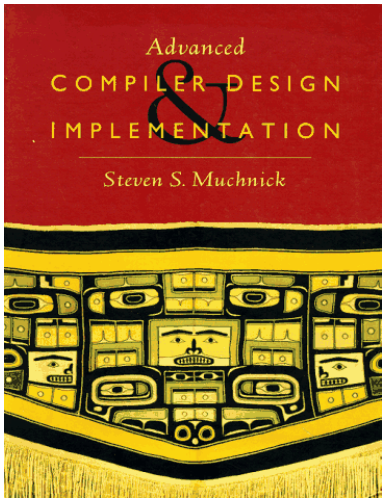
Andrew W. Appel:
Modern Compiler Implementation in Java
558 Seiten (12. März 1998)
Cambridge University Press; ISBN: 0521583888

Robert Morgan:
Building an Optimizing Compiler
300 Seiten (August 1997)
Butterworth-Heinemann; ISBN: 155558179X

Literatur – Hauptwerke II



Michael Joseph Wolfe, u.a.:
High Performance Compilers for Parallel Computing
570 Seiten (August 2000)
Addison-Wesley Pub Co; ISBN: 0805327304



Steve Muchnick:
Advanced Compiler Design and Implementation
888 Seiten (30. September 1997)
Morgan Kaufmann; ISBN: 1558603204

Literatur – Hauptwerke III

H.P. Zima (Editor):
Parallel Computation
First International ACPC Conference
Proceedings (LNCS 591)
Springer Verlag; ASIN: 0387554378

Cliff Click and Keith D. Cooper:
Combining Analyses, Combining Optimizations
ACM Transactions on Programming Languages and Systems
Bd. 17, No. 2, Seiten 181-196, 1995; ISSN: 0164-0925
www.acm.org/pubs/toc/Abstracts/0164-0925/201061.html

Literatur – Hauptwerke IV

Flemming Nielson, Hanne Riis Nielson, Chris Hankin:
Principles of Program Analysis
Springer Verlag 1999.
Corrected 2nd printing, 2005.
http://www2.imm.dtu.dk/~riis/PPA/ppa.html

Weitere Literatur I

R. W. Gray, Vince P. Heuring, S. P. Levi, A. M. Sloane, William M. Waite:
Eli: A Complete, Flexible Compiler Construction System
Comm. ACM, 1992, Bd. 35, No. 2, Seiten 121-131

G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, P. W. Markstein:
Register Allocation via Coloring
Journal of Computer Languages, 1981, Bd. 6, Seiten 45-57

Martin Trapp:
Optimierung Objekt-orientierter Programme
Universität Karlsruhe, Dez. 1999, Doktorarbeit

Markus Armbruster, Christian von Roques:
Entwurf und Implementierung eines Sather-K Übersetzers
Fakultät für Informatik, Universität Karlsruhe, Dez. 1996, Diplomarbeit

Weitere Literatur II

Sabine Glesner, Gerhard Goos, Wolf Zimmermann:
Verifix: Konstruktion und Architektur verifizierender Übersetzer
(Verifix: Construction and Architecture of Verifying Compilers)
it - Information Technology, 2004

Sebastian Hack, Daniel Grund, Gerhard Goos:
Register allocation for programs in SSA-form
Compiler Construction 2006, Springer, March 2006.
http://dx.doi.org/10.1007/11688839_20

Fundstellen für Übersetzerbaukästen im Netz

Übersichtsseiten zu Übersetzern und Übersetzerbaukästen:

- http://www.compilerconnection.com/
- http://www.compilers.net/index.htm
- http://compilers.iecc.com/tools.html
- http://wwwold.first.gmd.de/cogent/catalog/
- http://www.idiom.com/free-compilers/

Eine Archivseite der Newgroup „comp.compilers" mit Zusatzinfos

- http://compilers.iecc.com/ (comp.compilers archiv)

Software

Cocktail ein Übersetzerbaukasten:

- http://wwwold.first.gmd.de/cocktail/

ELI ein Übersetzerbaukasten mit Betonung des Erstellungsprozess:

- http://eli-project.sf.net

SUIF ein Übersetzerbaukasten mit besonderem Augenmerk auf Optimierungen:

- http://suif.stanford.edu/
- http://suif.stanford.edu/suif/suif2/index.html (neuere Version)
- http://www.eecs.harvard.edu/hube/research/machsuif.html (Machine SUIF)

Trimaran ein System zur Simulierung von Hardware für experimentelle Zwecke (insbesondere Parallelität auf Instruktionsebene)

- http://www.trimaran.org/

Open Research Compiler ORC (Intel beteiligt)

- "http://ipf-orc.sourceforge.net/"

Firm (auf Anfrage bei uns) implementiert SSA

PAG (Program Analyzer Generator), Uni Saarbrücken

Verifikation einer Übersetzung

Übersicht

- Korrektheit
- Spezifikation und Implementierung
- Kompositionalität
- Programmprüfung
- Zusammenfassung

Definition Quell-/Zielsprache

- Jede Korrektheitsaussage setzt eine präzise Definition der betrachteten Gegenstände, hier der beteiligten Sprachen voraus
- Quellsprache gegeben durch
  - Grammatik
  - Regeln der statischen Semantik
  - formale Spezifikation der dynamischen Semantik
- Zielsprache (Maschinensprache) gegeben durch
  - Maschinenbeschreibung (Wortgrößen, Speicher, Register, ...)
  - formale Spezifikation der Maschinenbefehle
  - beides findet man in Hardware-Beschreibungen der Prozessoren

Definition formaler Semantik

- Algebraische Spezifikationen,
- denotationelle Semantik: Programm definiert Abbildungen,
- natürliche Semantik
- ...
- operationelle Semantik:
  - abstrakte Zustandsmaschinen (abstract state machines, ASM): Programm definiert Zustandsübergänge
  - ...

Für unsere Zwecke nur operationelle Beschreibungen brauchbar:

- sonst keine Semantik für endlos laufende Programme
- Beschreibung der Quell- und Zielsprache mit dem gleichen Verfahren

abstrakte Zustandsmaschinen

ASMs verallgemeinern endliche Zustandsmaschinen, d.h. endl. Automaten

- jeder Zustand q_t ist eine Algebra mit Wertemenge und Operationen
- durch einen Zustandsübergang $q_t \rightarrow q_{t+1}$ werden die Ergebnisse von Funktionsaufrufen verändert
  - q_{t+1} unterscheidet sich von q_t nur durch diese veränderten Funktionsergebnisse
- die Veränderungen können unter Bedingung stehen usw.

Beispiel: (`ct` bezeichnet den „Befehlszähler" `current-task`, nullstellige Fkt.)

```
if ct = while then
   if value(ct,condition) = true then ct := truetask(ct)
   else ct := falsetask(ct)
   endif
endif
```

Zustände

Sequentielles Programm durchläuft Zustände

$q_0\, q_1\, q_2\, ...$

- Zustand q_t besteht aus Objekten, die zum Zeitpunkt t existieren: $q_t = (q_t^0, q_t^1, ..., q_t^n)$
- Operation entspricht Übergang $q_t \rightarrow q_{t+1}$
- Zustandsmenge Q die während Programmlauf existieren kann: Zustandsraum eines Programms (abhängig von Programm & Eingaben)

Beobachtbare Zustände

beobachtbar:

- Anfangs- und Endzustand
- Ein- und Ausgaben
- Im beobachtbaren Zustand q nur bestimmte Objekte $\underline{q}$ beobachtbar: $\underline{q}$ Teilfolge von $q_t = (q_t^0, q_t^1, ..., q_t^n)$
- Berechnung nicht beobachtbarer Objekte unerheblich, wenn sie terminiert
- Zwischenzustände unerheblich, wenn es nicht unendlich viele sind

Korrekte Übersetzung

- Betrachte nur beobachtbare Zustände:
  - deterministische Programme: Folge beobachtbarer Zustände
  - indeterministische Programme: azyklischer Graph beobachtbarer Zustände,
    Programmausführung entspricht Pfad vom Anfangs- zum Endzustand im Graph
  - parallele oder konkurrente Programme: azyklischer Graph beobachtbarer Zustände
    Programmausführung wählt einen azyklischen Teilgraphen aus (hier nicht näher betrachtet)

Azyklischer Zustandsgraph

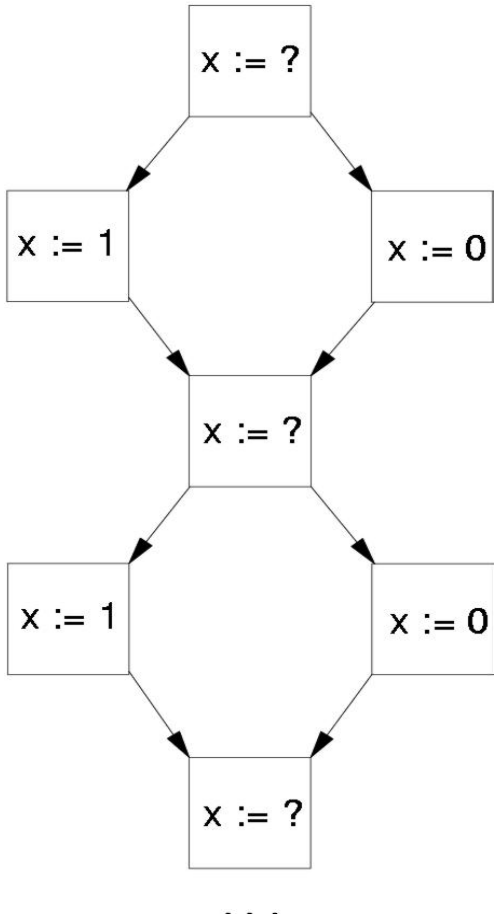
Indeterministisches Programm:

```
do
[]   true -> x:=1
[]   true -> x:=0
od
```

Korrekte Ausführung auch:

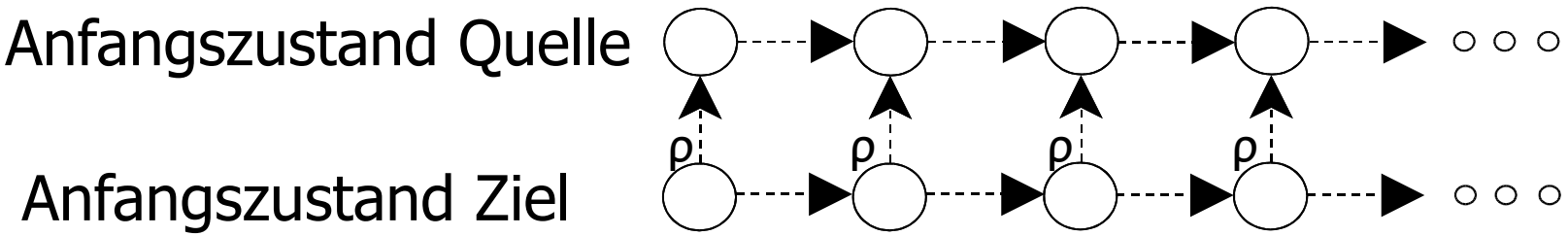
```
do true -> x:=1 od
```

Bei nichtdeterministischer Programmausführung darf sich der Übersetzer einen gültigen Pfad aussuchen.



Korrekte Übersetzung I

- Quellprogramm π
- Zielprogramm π'
- Bisimulation:
  Bei gleicher Eingabe gleiche beobachtbare Objekte in beobachtbarer Zustandsfolge
  (Relation ρ zwischen beobachtbaren Zuständen)



Korrekte Übersetzung präzise Definition

π′ ist korrekte Übersetzung von π, wenn für alle zulässigen Eingaben eine der folgenden Aussagen gilt:

- Zu jeder Folge beobachtbarer Zustände q_0', q_1', …, q_k' von π′ die regulär terminiert, gibt es eine terminierende Zustandsfolge q_0, q_1, …, q_k von π mit
  q_i ρ q_i' für $0 \le i \le k$;
- Zu jeder nicht-terminierenden Folge beobachtbarer Zustände q_0', q_1', …, von π′ gibt es eine Zustandsfolge q_0, q_1, …, von π mit q_i ρ q_i' für alle i;
- Zu jeder Folge beobachtbarer Zustände q_0', q_1', …, q_{k+1}' von π′, die mit einer Fehlermeldung wegen Verletzung von Betriebsmittelbeschränkungen terminiert, gibt es eine Zustandsfolge q_0, q_1, …, q_k, …von π mit q_i ρ q_i' für $0 \le i \le k$;

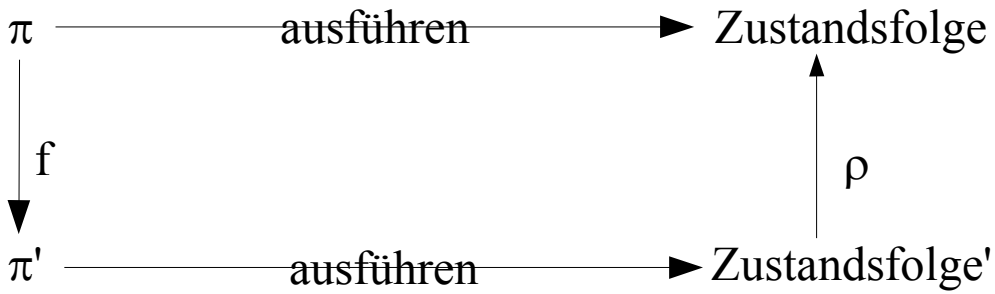
regulär: endliche viele Schritte ohne Fehlermeldung

ρ: korrespondierende beobachtbare Zustandsvariable haben gleichen Wert

Spezifikation und Implementierung

Korrektheitsbeweis in zwei Schritten:

- Ist die Spezifikation korrekt? Kommutiert das Diagramm



- Ist die Implementierung von f korrekt?

Kompositionalität

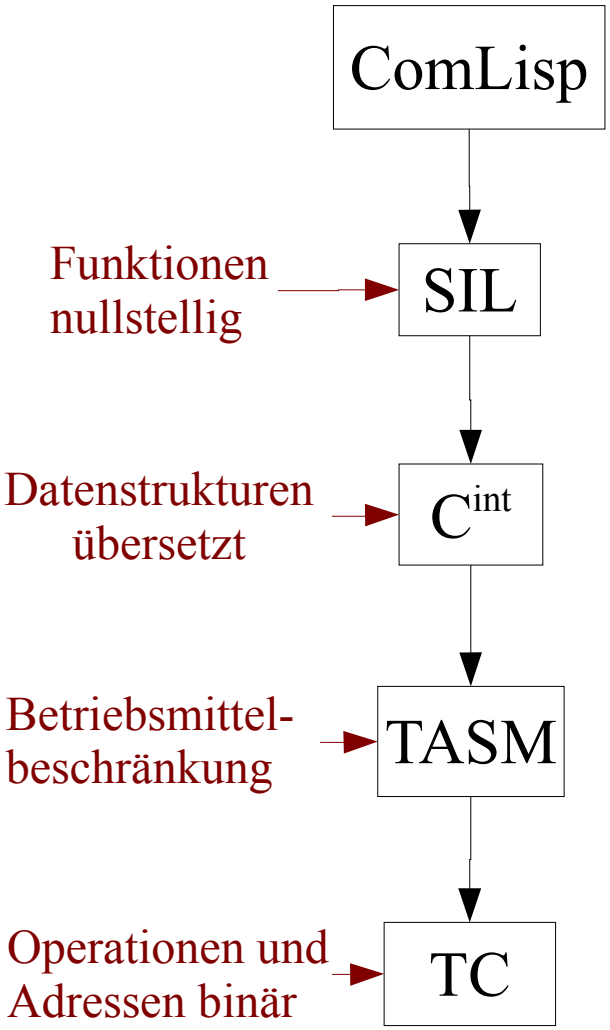
Sowohl die Spezifikationstreue als auch die Implementierung werden nicht in einem Schritt betrachtet

- horizontale Kompositionalität:
  - Semantik der Quellsprache entsprechend Syntaxbaum aus Semantik der Sprachelemente zusammengesetzt
  - analog: Semantik des Zielprogramms aus Abfolge der Befehle
- vertikale Kompositionalität:
  - Übersetzung nicht in einem Schritt, sondern mit zwischengeschalteten Zwischensprachen

Horizontale Kompositionalität folgt aus den Sprachdefinitionen.

Anwendbarkeit vertikaler Kompositionalität muß man (durch Bisimulation) beweisen.

Vertikale Kompositionalität im Verifix-Projekt



Teilmenge Common Lisp

stack intermediate language

Parameterübergabe und Ausdrucks-berechnung im Keller

ComLisp mit linearem Speicher

Datenstrukturen und Operatoren auf linearem Speicher abgebildet

symbolische Maschinensprache des Transputers T800

Registerkeller

binärer Transputercode

Implementierungskorrektheit

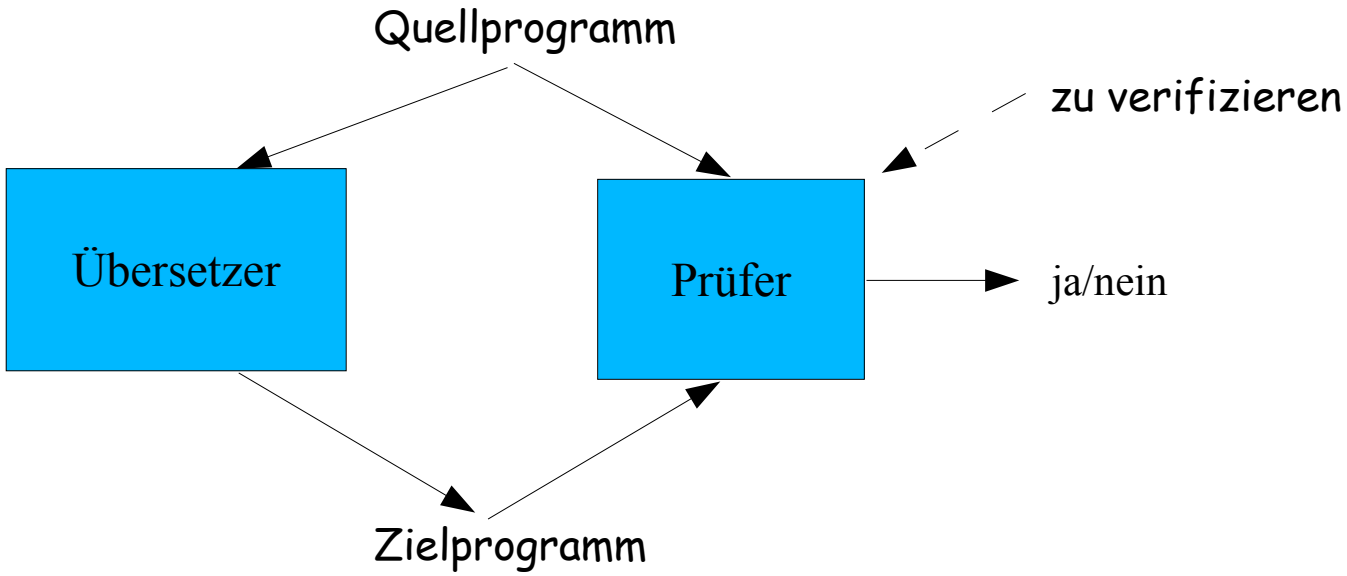
Nachweis durch korrekte Konstruktion

- schließt Verwendung von Generatoren aus
  (oder die Generatoren müßten verifiziert werden)
- Schwierigkeiten bei nicht-verfeinernden
  Optimierungstransformationen

also besser: Generatoren auf theoretischer Grundlage verwenden,
Implementierungskorrektheit auf andere Weise

Programmprüfung

Statt die Implementierung zu prüfen, prüfe nur das Ergebnis!



Methode von Blum&Kannan
„Programs that check their work"
auf Übersetzer angewandt von
Gaul & Goerigk

Prüfer garantiert
partielle Korrektheit,
aber: für jedes Programm
einzeln prüfen

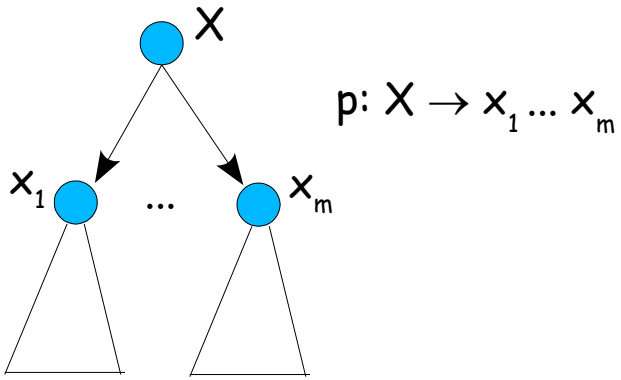
Programmprüfung II

Prüfung auch für Teilschritte geeignet, z.B.:

Prüfereingabe: kf. Grammatik G, Quellprogramm π,Syntaxbaum B

Prüferaufgabe: prüfe, daß der Baum B das Wort π korrekt wiedergibt,

- beginnend an der Baumwurzel, prüfe rekursiv, daß es eine passende Produktion p in G gibt
- Ausgabe „ja", wenn Syntaxbaum B korrekte Ableitung für π bzgl. G



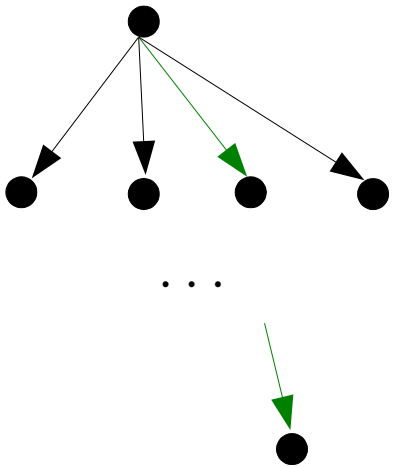
p: $X \rightarrow x_1 \ldots x_m$

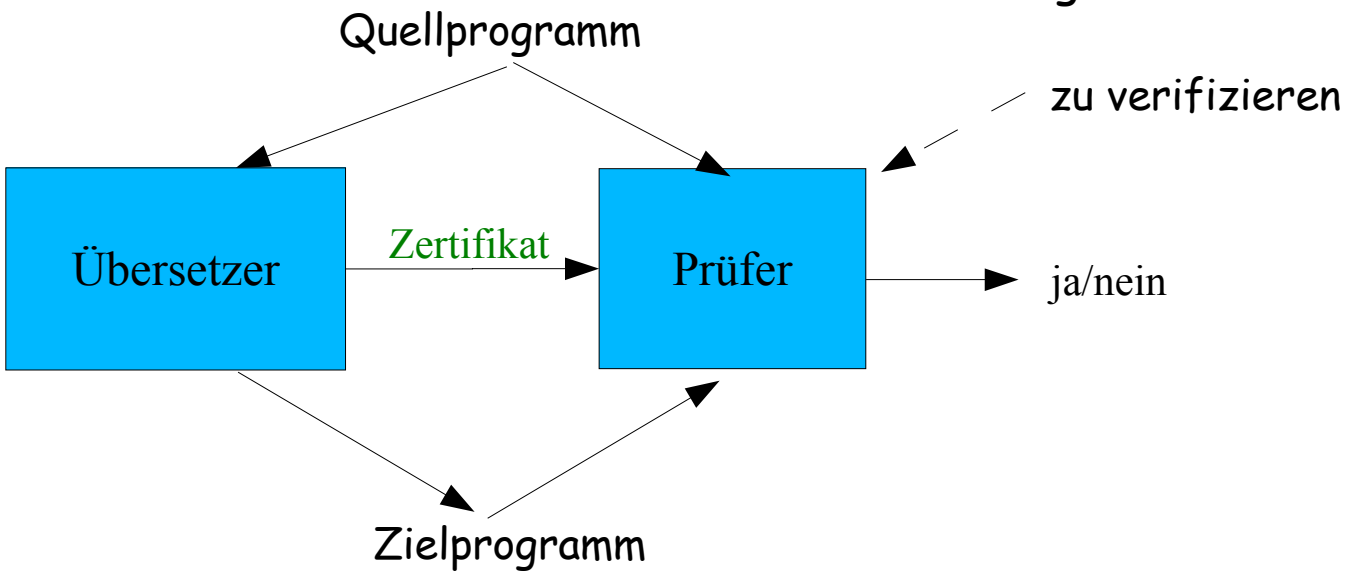
geeignet für alle Aufgaben der Analysephase

Programmprüfung III mit Zertifikaten

Erweiterung bei NP-Problemen und Problemen mit mehreren Lösungen

Bei NP-Problemen polynomiell viele Entscheidungsschritte

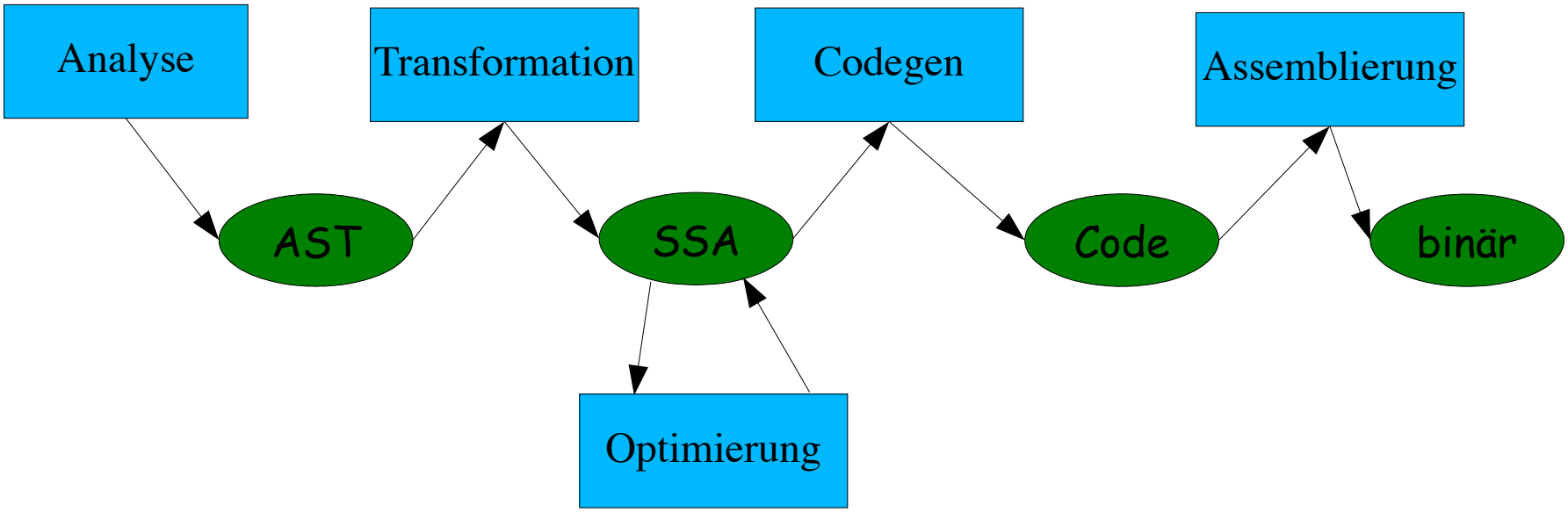




Weitergabe der Entscheidungsschritte als Zertifikat kann die Anzahl der „ja"-Antworten höchstens verkleinern

Aufgaben in der Synthesephase

- Korrektheit der SSA-Erzeugung (setzt formale Semantik für SSA voraus)
- Korrektheit der Optimierungen auf SSA-Form
- Korrektheit der Registerzuteilung
- Korrektheit der Codeerzeugung
- Korrektheit der Nachoptimierungen (Befehlsanordnung, usw.)
- Korrektheit der Assemblierung



Zusammenfassung

Gegenwärtiger Stand (Anfang 2006)

- Verifikationsprinzipien bekannt (direkter Beweis, Programmprüfung)
- Verfahren für einzelne Teilsprachen und Teilschritte durchgeführt
  - Comlisp, IS, Java-Ausschnitt, Sather-Ausschnitt
- maschinelle Verifikation mit Theorembeweiser noch in Arbeit (Glesner)
- noch keine vollständige Verifikation eines realistischen Übersetzers für eine realistische Sprache

Vergleich mit anderen Arbeiten:

- proof-carrying code (Necula) prüft Typsicherheit, nicht Korrektheit
- Optimierungsverifikation (Necula) prüft Teile der Korrektheit des gcc ohne Rückgriff auf Implementierungseigenschaften
- viele frühere Arbeiten