Kapitel 4
Korrektheit der Zwischencodeerzeugung

Wolf Zimmermann

Verifikation von Übersetzern

Inhalt

Ziele

- Funktionsweise von Simulationsbeweisen
- Verifikation von Transformationsregeln
- Konzept der Übersetzungsvalidierung

4.1 Einleitung

Aufgabe

Transformation des attributierten Strukturbaums in ein Zwischencodeprogramm

- Keine höheren Steuerstrukturen (außer evtl. meist parameterloser Prozeduraufrufe)
- ⇒ Nur noch bedingte und unbedingte Sprünge
- Ablaufmodell und Speicherorganisation bezieht sich auf Zielmaschine
- Adressrechnung ist explizit
- ⇒ Einbeziehung der Tabellen von Relativadressen
- ☞ Wir verwenden hier Registermaschinen

Registermaschinen

- Unbeschränkte Registerzahl
- Datentypen entsprechen denen der Zielmaschine
- Operationen entsprechen denen der Zielmaschine **aber keine konkreten Befehle**
- ⇒ Ausdrücke bleiben bis auf Funktionsaufrufe im Wesentlichen erhalten
- ⇒ Auf fast allen Prozessoren gleich

Sichten auf Zwischensprache

Menge von Grundblockgraphen

- Pro Prozedur ein Grundblockgraph
- Jeder Grundblock ist eine Folge von Befehlen, deren letzter Befehl ein bedingter oder unbedingter Sprung ist
- Grundblöcke sind Sprungziele.
- Es kann nur an den Anfang eines Grundblocks gesprungen werden, danach wird **jeder** Befehl im Grundblock nacheinander ausgeführt.

Folge von Tripeln und Quadrupeln

- t_1, t_2, t_3 notieren Werte (Register)
- Auch bedingte und unbedingte Sprünge
- Sprungziele sind Marken l
- Spezialfall: an jedes Register t_i wird im Programm nur einmal zugewiesen (**Static Single Assignment**, kurz: SSA)

Sichten auf Zwischensprache (Forts.)

Grundblock ist Folge von Quadrupeln

- Beginnt mit Marke $l : \cdots$ und endet mit bedingtem oder unbedingtem Sprung
- Sonst keine Sprünge oder Sprungziele
- 1-1-Korrespondenz zwischen Quadrupelfolge und Grundblockgraph
- Statt Quadrupel können auch direkt Ausdrücke verwendet werden

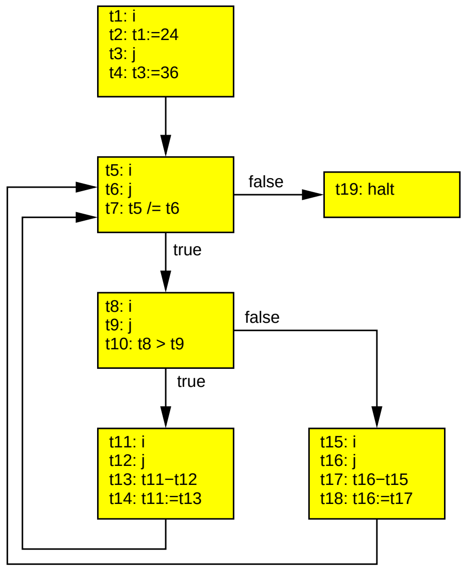
Datenflusssicht

- Kante von Definition von t zu Benutzung von t (**Def-Use-Kette**)
- Kante von Benutzung von t zu Definition von t (**Use-Def-Kette**)

⇒ Basis steckt bereits in Datenflusskanten

☞ Ggf. Verbesserung mit Datenflussanalyse

Beispiel 4.1: Sichten auf Zwischensprache



Grundblockgraph mit Tripelfolgen als Grundblöcken

$l_1 :$ $t_1 :=\uparrow i$
$(t_1) := 24$
$t_3 :=\uparrow j$
$(t_3) := 36$
goto l_2
$l_2 :$ $t_4 := i$
$t_5 := j$
if $t_4 \neq t_5$ **then** l_3 **else** l_4
l_3 $t_6 := i$
$t_7 := j$
if $t_6 > t_7$ **then** l_5 **else** l_6
$l_4 :$ **halt**
$l_5 :$ $t_8 := i$
$t_9 := j$
$t_{10} := t_8 - t_9$
$t_{11} :=\uparrow i$
$(t_{11}) := t_{10}$
goto l_2
$l_6 :$ $t_{12} := i$
$t_{13} := j$
$t_{14} := t_{13} - t_{12}$
$t_{15} :=\uparrow j$
$(t_{15}) := t_{14}$
goto l_2

Quadrupelliste

Diskussion

- Wechsel der Sicht je nach Aufgabe
- Transformation in Zwischencode: Folge von Tripeln pro Prozedur
- Optimierung: Grundblock- und Datenflussgraph
- Codeerzeugung:

⇒ Aus Sicht der Verifikation der Zwischencodeerzeugung ist die Tripel/-Quadrupelfolge geeignet

⇒ Aus Sicht der Verifikation ist Codeerzeugung ist der Grundblock als Folge von Ausdrücken und Zuweisungen geeignet

- Korrektheit der anderen Sichten kann durch Programmprüfung

Aufgaben der Zwischencodegenerierung

Abbildung der Operatoren

- Quellsprachoperatoren werden auf Zielsprachoperatoren abgebildet
- Ggf. Erzeugung automatischer Anpassungen
- Ggf. Verwendung von Algorithmen
- Erzeugung neuer Objekte

⇒ Transformation von Zuweisungen, Ausdrücken

Abbildung der Steuerstrukturen

- Abbildung auf Bedingte und unbedingte Sprünge
- Prozedur- und Funktionsaufrufe
- Prozedur- und Funktionsrückkehr
- Gehört eigentlich zu Ausdrücken
- Koroutinen und nebenläufige Prozesse

Verifikation

Simulationsbeweise, wobei ρ die Zustandspaare an den Grenzen des transformierten Programmkonstrukts in Beziehung setzt.

4.2 Eine Grundblockbasierte Zwischensprache

Überblick

- Programme sind Folge von Befehlen L : BEF, wobei die Sprungmarke L nicht notwendig ist
- Es gibt Lese- und Schreibbefehle, arithmetische Befehle (aber keine Ganzzahldivision bzw. –rest), Sprungbefehle, Befehle zum Sichern und Wiederherstellen von Zwischenergebnissen, Befehle zur Manipulation von Keller– und Haldenzeiger sowie Unterprogrammaufruf– und Rückkehr

Zustand

- Register speichern Zwischenwerte
- ☞ Unbegrenzte Resourcen
- Transformation ordnet jedem Ausdruck ein eindeutiges Register zu
- Speicher der DEC-Alpha samt Kellerzeiger, Haldenzeiger, *UP* und Statusregister
- Befehlszeiger enthält Befehlsnummer

Zwischensprachprogramme

spec $\textsc{Prog}_{IL}$ **extends** $\textsc{Value}_\alpha$,$\textsc{Int}$
sorts REGISTER, BEF, LOAD, LOADG, LOADA, STORE, STOREG, STOREA, INTPLUS, . . ., FLTPLUS, . . ., INTTOFLT, JMP, BEQ, . . ., SAVE, RESTORE, PUSH, POP, CALL, RETURN, ALLOC, BEFLIST, LABEL
subsorts LOAD $\sqsubseteq$ BEF, . . . , ALLOC $\sqsubseteq$ BEF
operations

mkload :	LABEL × REGISTER × REGISTER	→ LOAD	Laden aus Speicher
mkloada :	LABEL × REGISTER × QUAD	→ LOADA	Laden von Adresse bzgl. *BP*
mkloadg :	LABEL × REGISTER × QUAD	→ LOADG	Laden von Adresse bzgl. *UP*
mkstore :	LABEL × REGISTER × REGISTER	→ STORE	Speichern
mkconst :	LABEL × REGISTER × QUAD	→ CONST	Laden einer Konstante
$(\oplus_I)$:	LABEL × REGISTER^3	→ INTPLUS	Ganzzahladdition etc.
$(\oplus_F)$:	LABEL × REGISTER^3	→ FLTPLUS	Gleitkommaaddition etc.
$(=_I)$:	LABEL × REGISTER^3	→ INTEQ	Gleitkommaaddition etc.
$(=_F)$:	LABEL × REGISTER^3	→ FLTEQ	Gleitkommaaddition etc.
mkjmp :	LABEL × LABEL	→ JMP	unbedingter Sprünge
mkbeq :	LABEL × REGISTER × LABEL^2	→ BEQ	bedingter Sprung, Vergl. mit 0 etc.
mksave :	LABEL	→ SAVE	Sichern von Zwischenwerten
mkrestore :	LABEL	→ RESTORE	Wiederherstellen von Zwischenwerten
mkread :	LABEL × REGISTER	→ READ	*SP* erhöhen
mkwrite :	LABEL × REGISTER	→ WRITE	*SP* erhöhen
mkpush :	LABEL × QUAD	→ PUSH	*SP* erhöhen
mkpop :	LABEL × QUAD	→ POP	*SP* erniedrigen
mkcall :	LABEL × REGISTER × LABEL × QUAD	→ CALL	Prozeduraufruf
mkreturn :	LABEL × REGISTER	→ RETURN	Prozedurrückkehr
mkalloc :	LABEL × REGISTER × QUAD	→ ALLOC	*HP* erniedrigen
mknoreg :		→ REGISTER	
mkreg :	INT	→ REGISTER	
mknolab :		→ *LABEL*	
mklabel :	INT	→ *LABEL*	
getlab :	BEF	→ LABEL	Label des Befehls
labtobef :	BEFLIST × LABEL	→?INT	Befehl zu Label
getbef :	BEFLIST × INT	→ LABEL	*i*-ter Befehl
getreg :	LOAD	→ REGISTER	

und weitere Zugriffsbefehle

Zustandsraum und Anfangszustand

Zustandsraum

spec $\textsc{State}_{IL}$ **extends** $\textsc{Prog}_{IL}$
sorts BYTELIST
operations

mem_α :	QUAD	→ BYTE	
inp_α :		→ BYTELIST	
out_α :		→ BYTELIST	
$prog_i$:		→ BEFLIST	
ip :		→ INT	Befehlszeiger
fpcr :		→ QUAD	Statusregister
val :	REGISTER	→?QUAD	
UP :		→ QUAD	
BP :		→ QUAD	
SP :		→ QUAD	
HP :		→ QUAD	

Anfangszustand

spec $\textsc{InitState}_{IL}$ **extends** $\textsc{State}_{IL}$
axioms
$out_\alpha \doteq []$
$ip \doteq 0$
$fpcr \doteq O^{64}$
$BP \doteq UP$
$SP \doteq UP$

Beobachtung

Vor dem eigentlichen Start des Hauptprogramms muss Platz für die globalen Variablen geschaffen werden.

Laden und Speichern

Laden vom Speicher Relativadressen bzgl. *BP*

Notation: $t_i :=$ LD x

if *IP* **is** LOAD **then**
  $val(Reg) := Read(Opd_1)$
  $Proceed_i$

$IP \triangleq getbef(prog_i, ip)$
$Reg \triangleq getreg(IP)$
$Opd_j \triangleq opd_j(IP),\ j = 1, 2, 3$
$Read(l) \triangleq$ wie in Kapitel 3
$Proceed \triangleq ip := ip + 1$

Laden von Relativadressen bzgl. *UP*

Notation: $t_i :=$ LDG x

if *IP* **is** LOADG **then**
  $val(Reg) := UP \oplus_I Opd_1$
  $Proceed_i$

Laden von Relativadressen bzgl. *BP*

Notation: $t_i :=$ LDA t_j

if *IP* **is** LOADA **then**
  $val(Reg) := BP \oplus_I val(Opd_1)$
  $Proceed_i$

Speichern

Notation: $t_i :=$ ST t_h, t_j

$Store(l) \triangleq$ wie in Kapitel 3

if *IP* **is** STORE **then**
  $Store(Opd_1, val(Opd_2))$
  $Proceed_i$

Arithmetische Befehle

Ganzzahloperationen

Notation: $t_i := t_j \oplus_I t_h$ / $t_i := t_j \otimes_I t_h$ / $t_i := t_j \ominus_I t_h$

if IP **is** INTPLUS **then**
$val(Reg) := val(Opd_1) +_I val(Opd_2)$
$Proceed_i$

Gleitkommaoperationen

Notation: $t_i := t_j \oplus_F t_h$ / $t_i := t_j \otimes_F t_h$ / $t_i := t_j \ominus_F t_h$ / $t_i := t_j \oslash_F t_h$

if IP **is** FLTDIV **then**
if $val(Opd_2) \doteq 0$ **then**
$fpcr := LO^9 LO^{53}$
$ip := -1$
else $val(Reg) := val(Opd_1) /_F val(Opd_2)$
$Proceed_i$

Konversion

Notation: $t_i :=$ INTTOFLT t_j

if IP **is** INTTOFLOAT **then**
$val(Reg) := convert(val(Opd_1))$
$Proceed_i$

Vergleichsoperationen

Vergleich auf ganzen Zahlen

Notation: $t_i : t_i \leq_I t_j$
Analog: $=_I$, $<_I$

if IP **is** INTLEQ **then**
if $val(Opd_1) \leq_I val(Opd_1) \doteq true$ **then**
$val(Reg) := O^{63}L$
else $val(Reg) := O^{64}$
$Proceed_i$

Vergleich auf Gleitkommazahlen

Notation: $t_i : t_i \leq_F t_j$
Analog: $=_F$, $<_F$

if IP **is** FLTLEQ **then**
if $val(Opd_1) \leq_F val(Opd_1) \doteq true$ **then**
$val(Reg) := O^{63}L$
else $val(Reg) := O^{64}$
$Proceed_i$

Sprünge

Statische Semantik

- Im Programm gibt es keine zwei Befehle mit demselben Label, d.h. jeder Zustand erfüllt $\mathfrak{A} \models getlab(getbef(prog_i, j)) \doteq getlab(getbef(prog_i, k)) \Rightarrow i \doteq j \vee getlab(getbef(prog_i, j)) \doteq nolabel$
- Jedes Label in einem Sprungbefehl muss vorhanden sein.

$\Rightarrow$ Statische Funktion *labtobef* : BEFLIST $\times$ LABEL $\to$ *INT*

Unbedingter Sprung

Notation: JMP L
$LabToBef(l) \triangleq labtobef(prog_i, l)$

if IP **is** JMP **then**
$ip := LabToBef(Opd_1)$

Bedingte Sprünge

Notation:
BEQ t_j, L_1, L_2 BNE t_j, L_1, L_2
BGT t_j, L_1, L_2 BLT t_j, L_1, L_2
BGEQ t_j, L_1, L_2 BLEQ t_j, L_1, L_2

if IP **is** BEQ **then**
if $val(Opd_1) \doteq 0$ **then**
$ip := LabToBef(Opd_2)$
else $ip := LabToBef(Opd_3)$

Lesen und Schreiben

Lesen

Notation: READ t_j

if IP **is** READ **then**
$Store(Opd_1, head(inp_\alpha))$
$inp_\alpha := tail(inp_\alpha)$
$ip := ip + 1$

Schreiben

Notation: WRITE t_j

if IP **is** WRITE **then**
$out_\alpha := out_\alpha{+}{+}val(Opd_1)$
$ip := ip + 1$

Sichern und Wiederherstellen von Zwischenwerten

Statische Semantik

- Jedes geschriebene Register wird bis zum nächsten bedingten und unbedingten Sprung gelesen
- Zwischen dem SAVE und dem darauffolgenden RESTORE werden keine Register geschrieben

⇒ Alle Register mit lebendigen Werten können gesichert und wiederhergestellt werden

⇒ Statische Funktion livevals : BEFLIST × INT → REGISTERLIST liefert nach Registernummern sortierte Liste

Sichern der lebendigen Register

Notation: SAVE

$LiveVals \triangleq livevals(prog_i, ip)$
$Reg(i) \triangleq get(LiveVals, i) \doteq true$
$Rel(a, j) \triangleq a +_I inttoquad(j)$
$SizeLive \triangleq inttoquad(length(LiveVals) * 16)$

if IP **is** SAVE **then**
  forall i : INT • $0 \le i < length(LiveVals)$ **do**
    $Store(Rel(SP, 16 * i), inttoquad(i))$
    $Store(Rel(SP, 16 * i + 8), val(Reg(i)))$
  $SP := SP +_I SizeLive$
  $Proceed_i$

Wiederherstellen der lebendigen Register

Notation: RESTORE

$NewSP \triangleq SP -_I SizeLive$

if IP **is** RESTORE **then**
  forall i : INT • $0 \le i < length(LiveVals)$ **do**
    $val(Reg(i)) := Read(NewSP, 16 * i + 8)$
  $SP := NewSP$
  $Proceed_i$

Manipulation des Kellerzeigers und Prozeduraufrufe

Kellerzeiger erhöhen oder erniedrigen

Statische Semantik: Das Argument ist immer größer als 0

Notationen: PUSH n bzw. POP n

if IP **is** PUSH **then**
  if $SP +_I Opd_1 \ge_I HP \doteq true$ **then**
    $fpcr := LO^{63}$
    $ip := -1$
  else $SP := SP +_I Opd_1$
    $Procceed_i$

if IP **is** POP **then**
  $SP := SP -_I Opd_1$
  $Procceed_i$

Prozeduraufruf und -rückkehr

Idee:

- Bei Aufruf Umsetzen des Basisregistern, Sichern des alten Basisregisters und Ausführen des Unterprogramms
- Bei Rückkehr Wiederherstellen des alten Basisregisters, Funktionsergebnisse übergeben, und fortfahren mit dem nächsten Befehl.

Notationen: t_i : CALL L, x und RET

if IP **is** CALL **then**
  $BP := BP +_I Opd_2$
  $Store(BP +_I Opd_2, BP)$
  $Store(Rel(BP +_I Opd_2, 8), inttoquad(ip))$
  $ip := LabToBef(Opd_1)$

if IP **is** RET **then**
  $BP := Read(BP)$
  $ip := OldIP$
  $val(getreg(OldIP)) := val(Opd_1)$
  $ip := OldIP + 1$

$OldIP \triangleq quadtoint(Read(Rel(BP, 8)))$

Allokieren von Speicherplatz auf der Halde

Notation: $t_i :=$ ALLOC n

if IP **is** ALLOC **then**
  if $HP -_I Opd_1 < SP \doteq true$ **then**
    $fpcr := LO^{63}$
    $ip := -1$
  else $HP := HP -_I Opd_1$
    $Reg := HP -_I Opd_1$
    $Proceed_i$

4.3 Baumersetzungssyteme

Beobachtung

Die Zwischencodeerzeugung transformiert den attributierten Strukturbaum. Die durchzuführende Aktionen hängen ab

- von Mustern der abstrakten Syntax
- von Attributen, die in einem passenden Muster der abstrakten Syntax vorkommen (hier formalisiert durch statische Funktionen)

⇒ Spezifikation durch Transformationsregeln

☞ Hier sind die Aktionen die Ausgabe des zu erzeugenden Codes (Aufbauende Operationen, textuelle Ausgabe)

Muster

Ein **Muster** ist ein Term, der Variablen enthalten kann, z.B.

- $plus(X, Y)$ repräsentiert alle abstrakten Syntaxbäume mit Wurzel vom Typ *plus* ist
- X und Y sind die beiden Teilbäume
- *expr* repräsentiert alle abstrakten Syntaxbäume, deren Wurzel vom Typ *expr* ist
- Auch alle Untertypen

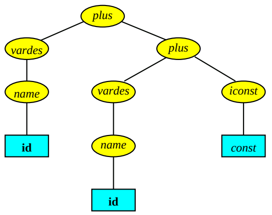
Muster und Abstrakte Syntaxbäume

Abstrakter Syntaxbaum T passt auf Muster M

- Jeder Syntaxbaum T passt auf Variable X
- Jeder Syntaxbaum T, dessen Wurzel Typ *nt* hat, passt auf Muster *nt*
- Jeder Syntaxbaum T mit Unterbäumen $T_1, \ldots, T_n$ passt auf Muster $nt(m_1, \ldots, m_n)$ wenn die Wurzel von T Typ *nt* hat und die Kinder T_i auf Muster m_i passen, $i = 1, \ldots, n$

Ein Muster m_1 ist **spezieller als Muster** m_2 gdw. jeder abstrakte Syntaxbaum T, der auf m_1 passt, auch auf m_2 passt.

Beispiel 4.1: Passende Baummuster



passt auf *expr*
passt auf $plus(X, Y)$
passt auf $plus(X, plus(Y, Z))$
passt auf $plus(vardes, X)$
passt auf $plus(vardes(X), plus(Y, Z))$

$plus(X, plus(Y, Z))$ ist spezieller als $plus(X, Y)$
plus ist spezieller als *expr*

Baumersetzungssregeln

Baumersetzungsregel

Eine **Baumersetzungsregel** hat die Form
$$\begin{array}{lll} \mathcal{T}[\![m]\!] & = Aktionen_1 & \textbf{if } Bedg_1 \\ & = Aktionen_2 & \textbf{if } Bedg_2 \\ & \ldots & \\ & = Aktionen_k & \textbf{if } Bedg_k \end{array}$$

- $\mathcal{T}$ ist der **Name** der Transformation
- m ist ein Muster
- $Aktionen_i$ können weitere Transformationen aufrufen und insbesondere Teile des Muster enthalten und Aufrufe statischer Funktionen enthalten
- $Bedg_i$ sind Formeln über der Signatur der Spezifikation $\textsc{Prog}$
- Die Navigationsliste *this* in diesen Formeln bezieht sich auf die Wurzel des Teilbaums, auf das den die Regel angewendet wird.

Anwendung einer Baumersetzungsregel

- Prüfe, ob Muster auf abstrakten Syntaxbaum passt
- Wähle erste erfüllte Bedingung
- Führe die entsprechenden Aktionen aus
- Falls Muster nicht passt oder keine der Bedingungen erfüllt ist, dann ist die Regel nicht **anwendbar**

Baumersetzungssystem

Baumersetzungssystem

Ein **Baumersetzungssystem** ist eine Menge von Baumersetzungsregeln.

Anwendung eines Baumersetzungssystem

- Finde anwendbare Regel mit speziellstem Muster, das noch auf den abstrakten Syntaxbaum passt
- Aktionen wenden ggf. Baumersetzungssystem auf Unterbäume an
- Solange durchführen, bis Baumersetzungssystem nicht mehr auf Unterbäume angewendet wird oder keine Regel mehr anwendbar ist

Ziel

Baumersetzungssystem soll so konstruiert sein, dass alle Produktionen der abstrakten Syntax abgedeckt sind

⇒ Kein Abbruch, weil keine Regel mehr anwendbar ist
⇒ Zwischencode wird komplett erzeugt
- Kann an Hand der Grammatik und an Hand der Regelmenge automatisch überprüft werden
- Es dürfen aber auch komplexere verschachtelte Muster verwendet werden.
- Werkzeuge generieren Code aus Baumersetzungssystemen

☞ Wir verwenden hier den Korrektheitsnachweis für Baumersetzungsregeln

4.4 Verifikation der Zwischencodeerzeugung

Vorgehen

- Aufstellen einer Transformationsregel
- Formulieren der notwendigen statischen Funktionen
- Korrektheitsnachweis durch Simulation

***n*-*m*-Simulation**

Dazu muss eine Relation zwischen den Zuständen der IL-ASM $\mathcal{Z}$ und der DEC-Alpha ASM $\mathcal{D}$ für `C--` definiert werden.

- *val* und *loc* existieren nicht mehr.
- *pc* ist durch *ip* ersetzt worden
- Das Zwischenprogramm $prog_I$ ist das Ergebnis der Transformation des `C--`-Programms *prog*
- Der restliche Zustandsraum ist identisch
- Die Ausdruckswerte und die Adressen sind nun über $val : REGISTER \to \text{QUAD}$ erfasst

⇒ Erweiterung der Spezifikation $\textsc{Prog}$ um die statische Funktionen $exprtoreg : \text{PROG} \times \text{OCC} \to ?\text{INT}$.

- Manche Auswertungen benötigen mehr als ein Register: Statische Funktion $numreg : \text{PROG} \times \text{OCC} \to ?\text{INT}$. Falls zwei Intervalle $[exprtoreg(p, o) - numreg(p, o) + 1, exprtoreg(p, o)]$ und $[exprtoreg(p, o') - numreg(p, o') + 1, exprtoreg(p, o')]$ nicht disjunkt sind, ist $o \doteq o'$

n-m-Simulation

Abbildung der Befehlszeiger auf Instruction Pointer

- Der Befehlszeiger *ip* entspricht *pc* an dem ersten Befehl des Unterbaums auf dem eine Transformationsregel angewendet wurde und dem nächsten Befehl
- Statische Funktione *applytrafo* : PROG × OCC → BOOL, die angibt, ob eine Baumersetzungsregel angewendet wurde
- Statische Funktion *mappc* : OCC →?INT, die angibt, dass aus der ersten Anweisung eines Unterbaums der i-te Befehl generiert wird.

Relation ρ

Sei $\mathfrak{Z}$ Zustand von $\mathcal{Z}$ und $\mathfrak{D}$ Zustand von $\mathcal{D}$. Wir definieren $(\mathfrak{Z},\mathfrak{D}) \in \rho$ gdw.

$$
\begin{array}{lll}
[\![mem_\alpha]\!]_Z & = & [\![mem_\alpha]\!]_D \\
[\![inp_\alpha]\!]_Z & = & [\![inp_\alpha]\!]_D \\
[\![out_\alpha]\!]_Z & = & [\![out_\alpha]\!]_D \\
[\![fpcr]\!]_Z & = & [\![fpcr]\!]_D \\
[\![UP]\!]_Z & = & [\![UP]\!]_D \\
[\![BP]\!]_Z & = & [\![BP]\!]_D \\
[\![SP]\!]_Z & = & [\![SP]\!]_D \\
[\![HP]\!]_Z & = & [\![HP]\!]_D \\
[\![prog_i]\!]_Z & = & compile([\![prog]\!]_D) \\
[\![val]\!]_D(o) & = & [\![val]\!]_Z([\![mkreg]\!]_Z([\![exprtoreg(prog,o)]\!]_D) \\
[\![loc]\!]_D(o) & = & [\![val]\!]_Z([\![mkreg]\!]_Z([\![exprtoreg(prog,o)-1]\!]_D) \\
[\![pc]\!]_D & = & [\![first(occ(prog,pc))]\!]_Z \text{ für ein } o \text{ mit} \\
 & & \quad [\![applytrafo(prog,o)]\!]_D \doteq true\text{, oder} \\
[\![pc]\!]_D & = & [\![next(prog,pc)]\!]_Z \text{ für ein solches } o \\
[\![mappc(prog,pc)]\!]_D & = & [\![ip]\!]_Z
\end{array}
$$

Auswertung von Zugriffspfaden

Transformationen

- $\mathcal{D}[\![des]\!]$ berechnet Adresse des Zugriffspfads
- $\mathcal{E}[\![expr]\!]$ berechnet (unangepassten) Wert von Ausdrücken

Hilfsinformation zur Definition der Transformationen

- *isStatic* : PROG × OCC → BOOL gibt an, ob das Objekt auf dem Laufzeitkeller liegt

Transformation $\mathcal{E}$

$$
\begin{array}{lll}
\mathcal{E}[\![des]\!] & = & \mathcal{D}[\![des]\!] \qquad\qquad \textbf{if } IsAtomic(this) \wedge Pri(this) \doteq Post(this) \\
 & & Val(this) := \text{LD } addr(RelAddrTab, Id(this)) \\
 & = & \mathcal{D}[\![des]\!] \qquad\qquad \textbf{if } IsAtomic(this) \\
 & & mkreg(Num) := \text{LD } addr(RelAddrTab, Id(this))
\end{array}
$$

$$
\begin{array}{ll}
IsStatic(o) & \triangleq isStatic(prog,o) \\
IsAtomic(o) & \triangleq isAtomic(Pri(o)) \\
IsGlobal(o) & \triangleq isGlobal(prog,o) \\
Val(o) & \triangleq mkreg(exprtoreg(prog,o)) \\
Loc(o) & \triangleq mkreg(exprtoreg(prog,o)-1) \\
Num & \triangleq numregs(prog, this
\end{array}
$$

Transformation für Variablenzugriff

$$
\begin{array}{lll}
\mathcal{D}[\![des]\!] = & Loc(this) := \text{LDA } addr(RelAddrTab, Id(this)) & \textbf{if } \neg IsGlobal(this) \\
 & Loc(this) := \text{LDG } addr(GlobAddrTab, Id(this)) & \textbf{if } IsGlobal(this)
\end{array}
$$

Korrektheit

Lemma 4.1 (n-m-Simulation für Variablenzugriff)

Mit ρ, *des* und $\mathcal{D}[\![des]\!]$ wird eine n-m-Simulation definiert, falls *des* nur als Zugriffspfad ausgewertet wird.

Beweis

Sei $(\mathfrak{Z},\mathfrak{D}) \in \rho$ ein Zustand mit $\mathfrak{D} \models CT$ **is** DES. Nach Voraussetzung gilt $\mathfrak{D} \models isExpr(prog,pc) \doteq false$

1. Fall: $\mathfrak{D} \models isGlobal(prog, Id) \doteq false$. Dann gilt

$Update(\mathfrak{D}) = \{\ loc(pc) := BP +_I addr(RelAddrTab, Id),\ pc := next(prog,pc)\}$ und $\mathcal{D}[\![des]\!] = Loc(pc) := \text{LDA } addr(RelAddrTab, Id)$. Somit ist $Update(\mathfrak{Z}) = \{\ val(exprtoreg(prog,pc)-1) := BP +_I addr(RelAddrTab, Id),\ ip := ip+1\}$

Damit gilt auch für die Nachfolgezustände $\mathfrak{Z}'$, $\mathfrak{D}'$:

$$
\begin{array}{ll}
[\![loc]\!]'_D([\![pc]\!]_D) & = [\![BP]\!]_D + [\![addr(RelAddrTab, Id)]\!]_D \\
 & = [\![BP]\!]_Z + [\![addr(RelAddrTab, Id)]\!]_Z
\end{array}
$$

Außerdem gilt:

$$
\begin{array}{ll}
[\![val]\!]_{Z'}([\![exprtoreg]\!]_D([\![prog]\!]_D, [\![pc]\!]_Z)-1) & = [\![BP]\!]_Z + [\![addr(RelAddrTab, Id)]\!]_Z \\
 & = [\![loc]\!]'_D([\![pc]\!]_D)
\end{array}
$$

Außerdem gilt $[\![mappc(next(prog,pc))]\!]_Z = [\![ip+1]\!]_D$. Somit gilt auch $[\![mappc(prog,pc)]\!]_{D'} = [\![ip]\!]_{Z'}$. Also gilt in diesem Fall $(\mathfrak{Z}',\mathfrak{D}') \in \rho$

2. Fall $\mathfrak{D} \models isGlobal(prog, Id) \doteq true$. **Übung**

Verbundfeld– und Klassenzugriffe

Transformation

$$
\begin{array}{lll}
\mathcal{D}[\![field(des, \textbf{id}))]\!] & = & \mathcal{D}[\![des]\!] \qquad\qquad \textbf{if } \neg IsClass(this) \\
 & & Aux(3) := Loc(Des(this)) \\
 & & Aux(2) := FieldSize(RelAddrTab, Pri(Des(this))) \\
 & & Loc(this) := Aux(3) \oplus Aux(2) \\
 & = & \mathcal{D}[\![des]\!] \\
 & & Aux(4) := Loc(Des(this)) \\
 & & Aux(3) := LD\ Aux(4) \\
 & & Aux(2) := FieldSize(RelAddrTab, Pri(Des(this))) \\
 & & Loc(this) := Aux(3) \oplus Aux(2)
\end{array}
$$

$$
\begin{array}{ll}
Des(o) & \triangleq des(prog,o) \\
Aux(i) & \triangleq mkreg(exprtoreg(prog, this) - i)
\end{array}
$$

Lemma 4.2 (n-m-Simulation für Feldzugriff)

Mit ρ, *field* und $\mathcal{D}[\![field]\!]$ wird eine n-m-Simulation definiert, falls *field* nur als Zugriffspfad ausgewertet wird.

Beweisidee

Sei $(\mathfrak{Z},\mathfrak{D}) \in \rho$, so dass $\mathfrak{D} \models pc \doteq first(prog, this)$. Falls durch $\mathcal{D}[\![Des(this)]\!]$ ein Zustand $\mathfrak{Z}'$ erreicht wird,f olgt durch Induktion, dass nach $Des(this)$ ein Zustand $\mathfrak{D}'$ mit $(\mathfrak{Z}',\mathfrak{D}') \in \rho$. Ausgehend von diesem Zustand beweist man dann die Korrektheit analog.

Ausdrücke

Erzeugung expliziter Anpassungen

Prinzip: Erst Ausrechnen, dann anpassen.
$\Rightarrow$ Transformation $\mathcal{E}'$, die ggf. Anpassung einführt

$$\begin{array}{lll} \mathcal{E}'[\![expr]\!] & = \mathcal{E}[\![expr]\!] & \textbf{if } Pri(this) \doteq Post(this) \\ & = \mathcal{E}[\![expr]\!] & \\ & \quad Val(this) := \text{INTTOFLOAT } Aux(Num) & \end{array}$$

Lemma 4.3 (Korrektheit der Auswertung der Zugriffspfade)

i. Mit ρ, Variablenzugriff *des* und $\mathcal{E}'[\![des]\!]$ wird eine *n*-*m*-Simulation definiert, falls *des* als Ausdruck ausgewertet wird.
ii. Mit ρ, Variablenzugriff *field* und $\mathcal{E}'[\![des]\!]$ wird eine *n*-*m*-Simulation definiert, falls *field* als Ausdruck ausgewertet wird.

Beweisskizze

Es gilt $\mathfrak{D} \models isExpr(prog, pc) \doteq true$
i.
1. Fall $\mathfrak{D} \models Pri(pc) \doteq Post(pc)$. Dann ist
$\mathcal{E}'[\![des]\!] = Loc(pc) :=$ LDA $addr(RelAddrTab, Id)$; Val$(pc) :=$ LD $Loc(pc)$. Somit ist
$Update(\mathfrak{Z}) = \{\ val(exprtoreg(prog, pc) - 1) := BP +_I addr(RelAddrTab, Id),\ ip := ip + 1\}$ und
$Update(next_{Update}\mathfrak{Z}) =$
$\{\ val(exprtoreg(prog, pc)) := Read(val(exprtoreg(prog, pc)) - 1), ip := ip + 1\}$ **Übung**
2. Fall: $\mathfrak{D} \models \neg Pri(pc) \doteq Post(pc)$. Dann ist $\mathfrak{D} \models Pri(pc) \doteq inttype$ und $\mathfrak{D} \models Post(pc) \doteq flttype$.
Übung

Ausdrucksauswertung: Arithmetische Opertionen

Addition

$$\begin{array}{lll} \mathcal{E}[\![plus(expr_1, expr_2)]\!] & = \mathcal{E}'[\![expr_1]\!] & \textbf{if } Pri(this) \doteq inttype \\ & \quad \mathcal{E}'[\![expr_2]\!] & \\ & \quad Val(this) := Val(Lhs) \oplus_I Val(Rhs) & \\ & = \mathcal{E}'[\![expr_1]\!] & \textbf{if } Pri(this) \doteq flttype \\ & \quad \mathcal{E}'[\![expr_2]\!] & \\ & \quad Val(this) := Val(Lhs) \oplus_F Val(Rhs) & \end{array}$$

$Lhs \triangleq lop(prog, this)$
$Rhs \triangleq rop(prog, this)$

Lemma 4.3 (Korrektheit der Addition)

Mit ρ, Addition *plus* und $\mathcal{E}'[\![\cdot]\!]$ wird eine *n*-*m*-Simulation definiert.

Ausdrucksauswertung: Division

Prinzip: Es muss eine in der Zwischensprache implementierte Hilfsfunktion aufgerufen werden.

$$\begin{array}{lll} \mathcal{E}[\![div(expr_1, expr_2)]\!] & = \mathcal{E}'[\![expr_1]\!] & \textbf{if } Pri(this) \doteq inttype \\ & \quad \mathcal{E}'[\![expr_2]\!] & \\ & \quad \text{SAVE} & \\ & \quad \text{PUSH } 32 & \\ & \quad Aux(2) := \text{LDA } (SizeLive(this) +_I SizeLocals(this) + 16) & \\ & \quad Aux(1) := Aux(2) \oplus 8 & \\ & \quad ST\ \ Aux(2), Val(Lhs) & \\ & \quad ST\ \ Aux(1), Val(Rhs) & \\ & \quad Val(this) := CALL\ divlab, SizeLive(this) +_I SizeLocals(this) & \\ & \quad \text{POP } SizeLive(this) +_I SizeLocals(this) +_I 16 & \\ & \quad \text{RESTORE} & \\ & = \mathcal{E}'[\![expr_1]\!] & \textbf{if } Pri(this) \doteq flttype \\ & \quad \mathcal{E}'[\![expr_2]\!] & \\ & \quad Val(this) := Val(Lhs) \oslash_F Val(Rhs) & \end{array}$$

$SizeLive(o) \triangleq inttoquad(length(livevals(prog, this)) * 16)$
$SizeLocals(o) \triangleq FrameSize(reladd(prog, o))$
divlab ist das Label, an dem die Divisionsprozedur beginnt.

Lemma 4.4 (Korrektheit der Divison)

Mit ρ, Addition *div* und $\mathcal{E}'[\![\cdot]\!]$ wird eine *n*-*m*-Simulation definiert, sofern in der Prozedur, die bei *divlab* beginnt, die Ganzzahldivision korrekt implementiert ist.

Boolesche Operatoren

Konjunktion

Prinzip: Kurzauswertung wird durch Sprung implementiert

- Statische Funktion *label* : PROG $\times$ OCC $\rightarrow$?INT, die eine Sprungmarke zuordnet

$\Rightarrow$ Die folgende Beweisverpflichtung (*) muss gelten:
$[\![labtobef]\!]_Z(compile([\![p]\!]_D), [\![mklab]\!]_D([\![label(p, o)]\!]_D)) = [\![mappc(p, o)]\!]_Z$

Transformation:

$$\begin{array}{lll} \mathcal{E}[\![and(expr_1, expr_2)]\!] = & & [\![expr_1]\!] \\ & & \text{BEQ } Val(Lhs), Lab(Lhs), Lab(Rhs) \\ & Lab(Lhs): & Val(this) := 0 \\ & & \text{JMP } Lab(Nxt(this)) \\ & Lab(Rhs): & [\![expr_2]\!] \\ & & Aux(1) := 0 \\ & & Val(this) := Val(Rhs) \oplus_I Aux(1) \\ & & \text{JMP } Lab(Nxt(this)) \end{array}$$

Makros: $Lab(o) \triangleq mklab(label(prog, o))$
$Nxt(o) \triangleq next(prog, o)$

Lemma 4.4 (Korrektheit der Konjunktion)

Mit ρ, Konjunktion *and* und $\mathcal{E}'[\![\cdot]\!]$ wird eine *n*-*m*-Simulation definiert, sofern die obige Beweisverpflichtung erfüllt ist.

Einführung von Sprüngen in Anweisungen

Grundidee

Man erzeugt zunächst die Sprungmarken für die Sprungziele (d.h. auf den Anfang von Grundblöcken) und unbedingte Sprünge am Ende der Grundblöcke

- ⇒ Transformation $\mathcal{S}'$ ohne Erzeugung von Sprungmarken und abschließenden Sprung und Transformation $\mathcal{S}$ mit abschließendem Sprung
- ⇒ Weitere statische Funktionen:

$bbBegin$:	PROG × OCC	→?BOOL	Grundblockanfang
$bbEnd$:	PROG × OCC	→?BOOL	Grundblockende
$target$:	PROG × OCC	→?INT	Sprungziel

- ⇒ Damit die Sprungstruktur korrekt abgebildet wird, muss gelten

$$\mathfrak{D} \models bbBegin(p, next(p, o)) \doteq bbEnd(p, o)$$
$$\mathfrak{D} \models bbEnd(p, o) \doteq true \Rightarrow target(p, o) \doteq label(p, next(p, o))$$

Transformation

$$\begin{array}{lll} \mathcal{S}[\![stat]\!] & = \mathcal{S}'[\![stat]\!] & \textbf{if } \neg BBBegin(this) \wedge \neg BBEnd(this) \\ & = Lab(this) : \mathcal{S}'[\![stat]\!] & \textbf{if } BBBegin(this) \wedge \neg BBEnd(this) \\ & = \mathcal{S}'[\![stat]\!] & \textbf{if } \neg BBBegin(this) \wedge BBEnd(this) \\ & \quad \text{JMP } Target(this) & \\ & = Lab(this) : \mathcal{S}'[\![stat]\!] & \textbf{if } BBBegin(this) \wedge BBEnd(this) \\ & \quad \text{JMP } Target(this) & \end{array}$$

Makros:

$$\begin{array}{ll} BBBegin(o) & \triangleq bbBegin(prog, o) \\ BBEnd(o) & \triangleq bbEnd(prog, o) \\ Target(o) & \triangleq mklab(target(prog, o)) \end{array}$$

Anweisungsfolgen

Leere Anweisungsfolge

$\mathcal{S}'[\![nostat]\!] =$

Nicht-Leere Anweisungsfolge

$$\mathcal{S}'[\![stats_1(stats_2, stat)]\!] = \begin{array}{l} \mathcal{S}[\![stats_2]\!] \\ \mathcal{S}[\![stat]\!] \end{array}$$

Lemma 4.5 (Korrektheit der Anweisungsfolgen)

Sei $\mathfrak{D}$ ein Zustand der DEC-Alpha-Semantik $\mathcal{D}$, so dass für alle Programme p und alle o mit $\mathfrak{D} \models o$ **is** STATS eine der beiden folgenden Eigenschaften erfüllt ist: $\mathfrak{D} \models occ(p, o.0) \doteq mknostat$
$\mathfrak{D} \models bbBegin(p, occ(p, o.1)) \doteq bbEnd(p, occ(p, o.0.1))$
Dann erfüllt die Transformation $\mathcal{S}$ zusammen mit ρ für Anweisungsfolgen die Bedingungen der n-m-Simulation

Transformation für Zuweisung

$$\begin{array}{lll} \mathcal{S}'[\![assign(des, expr)]\!] & = \mathcal{D}[\![des]\!] & \textbf{if } \text{isAtomic}(this.0) \\ & \quad \mathcal{E}'[\![expr]\!] & \\ & \quad \text{ST } Loc(this), Val(this) & \\ & = \mathcal{D}[\![des]\!] & \textbf{if } IsStruct(Pri(this.0)) \\ & \quad \mathcal{D}[\![expr]\!] & \\ & \quad \mathcal{A}[\![Fields(Pri(this.0))]\!]\ Loc(Strct(this.0))\ Loc(Field(this.0))\ FirstReg(this.0) & \end{array}$$

Makros:

$$\begin{array}{ll} NumReg(o) & \triangleq numreg(prog, o) \\ Strct(o) & \triangleq strct(prog, o) \\ DefTab(o) & \triangleq deftab(prog, o) \\ Def(o, x) & \triangleq identifyDef(DefTab(o), x) \\ IsStrct(x) & \triangleq isStruct(Def((), x)) \\ Fields(x) & \triangleq getFields(Def((), x)) \\ FirstReg(o) & \triangleq exprotreg(prog, this.0) - NumReg(this) + 1 \\ FldId(o) & \triangleq fldId(prog, o) \end{array}$$

Transformation für Feldweises Kopieren:

$$\begin{array}{lll} \mathcal{A}[\![field]\!]\ d\ e\ n & = mkreg(n) := reladdr(RelAddrTab(this), FldId(this)) & \textbf{if } isAtomic(Ty \\ & \quad mkreg(n+1) := d +_I mkreg(n) & \\ & \quad mkreg(n+2) := e +_I mkreg(n) & \\ & \quad mkreg(n+3) := \text{LD } mkreg(n+2) & \\ & \quad \text{ST } mkreg(n+1), mkreg(n+3) & \\ & = mkreg(n) := reladdr(RelAddrTab, FldId(this)) & \\ & \quad mkreg(n+1) := d +_I mkreg(n) & \\ & \quad mkreg(n+2) := e +_I mkreg(n) & \\ & \quad \mathcal{A}[\![Fields(\text{Type}(this.0))]\!]\ mkreg(n+1)\ mkreg(n+2)\ n+3 & \\ \mathcal{A}[\![fields(field)]\!]\ d\ e\ n & = \mathcal{A}[\![field]\!]\ d\ e\ n & \\ \mathcal{A}[\![fields_1(fields_2, field)]\!]\ d\ e\ n & = \mathcal{A}[\![fields_2]\!]\ d\ e\ n & \\ & \quad \mathcal{A}[\![field]\!]\ d\ e\ n + NumRegs(this.0) & \end{array}$$

Korrektheit der Zuweisung

Lemma 4.6 (Korrektheit der Zuweisung)

Mit ρ, Zuweisung *assign* und $\mathcal{S}[\![\cdot]\!]$ wird eine n-m-Simulation definiert.

Beweisidee

- Fallunterscheidung, ob Zuweisung am Ende eines Grundblock ist und damit ein Sprung generiert wird oder nicht.
  - ⇒ Zeigt, dass an den Zuständen $\mathfrak{D}'$ bzw. $\mathfrak{Z}'$ nach einer Zuweisung *assign* bzw. nach $[\![assign]\!]$ die Eigenschaft für die Befehlszeiger erfüllt ist, d.h. $[\![mappc(prog, pc)]\!]_{D'} = [\![ip]\!]_{Z'}$
- ⇒ Jetzt muss noch $[\![mem_\alpha]\!]_{Z'} = [\![mem_\alpha]\!]_{D'}$ gezeigt werden.
- Für atomare Werte kann das direkt gezeigt werden.
- Für Verbundkopien wird das durch Induktion gezeigt:
  - Wegen der statischen Semantik kommt man auf jeden Fall in einen Verbund, der nur noch Felder mit atomaren Werten enthält
  - Korrektheit für atomare Werte kann direkt gezeigt werden
  - Korrektheit für nichtatomare Felder folgt per Induktion
  - ⇒ Verbundobjekt ist kopiert, wenn die Verbundfelder sich nicht überlappen
  - Letzteres ist wegen der statischen Semantik erfüllt.

Transformation Ein-Ausgabe-Anweisungen

Leseanweisung

$$\begin{aligned}\mathcal{S}'[\![mkread(des)]\!] = \; & \mathcal{D}[\![des]\!]\\ & \text{READ } Opd_1\end{aligned}$$

Schreibeanweisung

$$\begin{aligned}\mathcal{S}'[\![mkwrite(expr)]\!] = \; & \mathcal{E}[\![expr]\!]\\ & \text{WRITE } val(Opd_1)\end{aligned}$$

Lemma 4.7 (Korrektheit der Lese- und Schreibeanweisung)

i. Mit ρ, der Leseanweisung *read* und $\mathcal{S}[\![\cdot]\!]$ wird eine n-m-Simulation definiert.

ii. Mit ρ, der Schreibeanweisung *write* und $\mathcal{S}[\![\cdot]\!]$ wird eine n-m-Simulation definiert.

Verzweigung

Transformation

Statische Semantik: Für alle Programme p und Navigationslisten o mit $\mathfrak{D} \models occ(p, o)$ **is** IF gilt:

$$\begin{aligned}\mathcal{D} &\models bbBegin(p, yes(p, o))\\ \mathcal{D} &\models bbBegin(p, no(p, o))\\ \mathcal{D} &\models bbBegin(p, next(p, o))\\ \mathcal{D} &\models bbEnd(p, o.1)\\ \mathcal{D} &\models bbEnd(p, o.2)\end{aligned}$$

Transformation:

$$\begin{aligned}\mathcal{S}'[\![if(expr, stat_1, stat_2)]\!] = \; & \mathcal{E}'[\![expr]\!]\\ & \text{BNE } val(Opd_1), Lab(Yes(this)), Lab(No(this))\\ & \mathcal{S}[\![stat_1]\!]\\ & \mathcal{S}[\![stat_2]\!]\end{aligned}$$

Makros:

$$\begin{aligned}Yes(o) &\triangleq yes(prog, o)\\ No(o) &\triangleq no(prog, o)\end{aligned}$$

Korrektheit der Verzweigung

Lemma 4.8 (Korrektheit der Verzweigung)

Mit ρ, der Verzweigunganweisung *if* und $\mathcal{S}[\![\cdot]\!]$ wird eine n-m-Simulation definiert, sofern die Bedingungen der statischen Semantik erfüllt sind.

Beweisidee

- Induktioves Argument für *expr*
- Direkter Simulationsnachweis für Zustandsübergänge von IF und bedingtem Sprung
- Wegen Bedingungen der statischen Semantik ist $\mathcal{S}[\![stat_i]\!] = \mathcal{S}'[\![stat_i]\!]$; JMP $Target(next(p, o))$

$\Rightarrow$ Induktives Argument gefolgt von Zustandsübergang durch Sprung

Schleifen

Statische Semantik: Für alle Programme p und Navigationslisten o mit $\mathfrak{D} \models occ(p, o)$ **is** WHILE gilt:

$$\begin{aligned}\mathcal{D} &\models bbBegin(p, yes(p, o))\\ \mathcal{D} &\models bbBegin(p, no(p, o))\\ \mathcal{D} &\models bbBegin(p, first(p, o))\\ \mathcal{D} &\models bbEnd(p, o)\end{aligned}$$

Transformation:

$$\begin{aligned}\mathcal{S}[\![while(expr, stat,]\!] = \; & Lab(Yes(this)): && \mathcal{S}[\![stat]\!]\\ & Lab(this): && \mathcal{E}[\![expr]\!]\\ & && \text{BNE } val(Opd_1), Lab(Yes(this)), Lab(No(this))\end{aligned}$$

Lemma 4.9 (Korrektheit der Schleife)

Mit ρ, der Schleifenanweisung *while* und $\mathcal{S}[\![\cdot]\!]$ wird eine n-m-Simulation definiert, sofern die Bedingungen der statischen Semantik erfüllt sind.

Beweisidee

- Aus statischer Semantik folgt $[\![mappc(prog, first(prog, o))]\!]_Z = [\![labtobef]\!]_I(compile([\![prog]\!]_Z), [\![Lab(o)]\!]_Z)$
- Induktives Argument für *expr*
- Direkte Simulation für WHILE-Anweisung und BNE-Befehl
- Fallunterscheidung über Ausgang der Bedingung
- Bei *false* ist alles erledigt
- Bei *true* Induktives Argument gefolgt vom Zustandsübergang durch Sprung

Schleifenabbruch und –fortsetung

Schleifenabbruch

Statische Semantik: Für alle Programme p und Navigationslisten o mit $\mathfrak{D} \models occ(o,p) \in$ BREAK gilt: $\mathfrak{D} \models target(p,o) \doteq label(break(p,o))$

Transformation: $\mathcal{S}'[\![break]\!] = \text{JMP } target(prog, this)$

Schleifenfortsetzung

Statische Semantik: Für alle Programme p und Navigationslisten o mit $\mathfrak{D} \models occ(o,p) \in$ CONTINUE gilt: $\mathfrak{D} \models target(p,o) \doteq label(continue(p,o))$

Transformation: $\mathcal{S}'[\![continue]\!] = \text{JMP } target(prog, this)$

Lemma 4.10 (Korrektheit des Break- und Continue-Anweisung)

i. Mit ρ, dem Schleifenabbruch *break* und $\mathcal{S}[\![\cdot]\!]$ wird eine n-m-Simulation definiert, sofern die Bedingungen der statischen Semantik erfüllt sind.

ii. Mit ρ, der Schleifenfortsetzung *continue* und $\mathcal{S}[\![\cdot]\!]$ wird eine n-m-Simulation definiert, sofern die Bedingungen der statischen Semantik erfüllt sind.

Beweisidee

Direkte Simulation

Blöcke

Transformation

$$\begin{aligned} \mathcal{S}'[\![block(decls, stats)]\!] \triangleq\ & \text{PUSH } SizeLocals(this) \\ & \mathcal{S}[\![stats]\!] \\ & \text{POP } SizeLocals(this) \end{aligned}$$

Lemma 4.11 (Korrektheit des Blocks)

Mit ρ, der Blockanweisung *block* und $\mathcal{S}[\![\cdot]\!]$ wird eine n-m-Simulation definiert.

Beweisidee

- Direkte Simulation für BLOCK-Anweisung bei Betreten
- Induktives Argument für *stats*
- Direkte Simulation für BLOCK-Anweisung bei Verlassen

Erzeugen von Objekten

Transformation und Makros

Transformation:

$\mathcal{E}'[\![new(type)]\!] = Loc(this) := \text{ALLOC } FieldsSize(this)$

Makro: $FieldsSize(o) \triangleq FrameSize(reladdr(prog, o.0))$

Lemma 4.12 (Korrektheit der Objekterzeugung)

Mit ρ, der Objekterzeugung *new* und $\mathcal{E}'[\![\cdot]\!]$ wird eine n-m-Simulation definiert.

Beweis

Direkte Simulation

Prozeduraufruf

Statische Semantik

- Jeder Rumpf einer Prozedur im Programm ist ein Grundblockanfang, d.h. falls $\mathfrak{D} \models occ(p,o)$**is** *PROCDECL* gilt: $\mathfrak{D} \models bbBegin(p, o.3)$ und hat damit ein Label.
- Jedem Argument in einem Prozeduraufruf ist eine Relativadresse *paraddr* : PROG $\times$ OCC $\to$?INT des entsprechenden Parameters der Prozedur zugeordnet, sowie die Größe *savesize* : PROG $\times$ OCC $\to$?INT der zu sichernden Ausdruckszwischenergebnisse gegeben. Außerdem gilt $\mathfrak{D} \models numregs(p,o) \doteq 4$ für alle p, o mit $\mathfrak{D} \models occ(p,o)$**is** *ARGS* (außer $\mathfrak{D} \models occ(p,o)$**is** *NOARGS*)
- Jeder Prozeduraufruf enthält ein Rückgaberegister der aufgerufenen Prozedur *retreg* : PROG $\times$ OCC $\to$?INT. Dieses Rückgaberegister ist allen Ausdrücken bei den Rückgabeanweisugen zugeordnet.

$\Rightarrow$ Beweisverpflichtungen für alle p, o mit $\mathfrak{D} \models occ(p,o)$ **is** CALL:

$$\begin{aligned} \mathfrak{D} &\models savesize(p, arg(p,o,i)) \doteq length(exprproc(pc)) * 16 + FrameSize(reladdr(p,o)) \\ \mathfrak{D} &\models paraddr(p, arg(p,o,i)) \doteq addr(Proc(p,o), get(pars(Proc(p,o), i))) \\ \mathfrak{D} &\models occ(p,o') \textbf{ is } \text{RETURN} \wedge InBody(Proc(p,o), o') \Rightarrow retreg(p,o) \doteq exprtoreg(p,o') \end{aligned}$$

Makro

$$\begin{aligned} Proc(o,i) &\triangleq identifyDef(DefTab(p,o), id(p, o.0)) \\ inBody(def, o) &\triangleq isPrefix(body(def), o') \end{aligned}$$

Prozeduraufruf

Transformation

$$\begin{array}{lll}
\mathcal{S}'[\![call(\mathbf{id}, args)]\!] & = & \text{SAVE} \\
 & & \text{PUSH } ParsSize(this) \\
 & & \mathcal{E}[\![args]\!] \\
 & & Val(this) : \text{CALL } ProcLab(this), SaveSize(this) \\
 & & \text{POP } ParsSize(this) \\
 & & \text{RESTORE} \\
\mathcal{E}[\![noargs]\!] & = & \\
\mathcal{E}[\![args_1(args_2, expr)]\!] & = & \mathcal{E}[\![args_2]\!] \qquad \textbf{if } isAtomic(this) \\
 & & Aux(3) := SaveSize(this) \\
 & & Aux(2) := ParAddr(this) \\
 & & Aux(1) := Aux(3) \oplus_i Aux(2) \\
 & & Aux(0) := \text{LDA } Aux(1) \\
 & & \mathcal{E}'[\![expr]\!] \\
 & & \text{ST}, Aux(0), Val(this.0) \\
 & = & \mathcal{E}[\![args_2]\!] \qquad \textbf{if } isStruct(Pri(this.1)) \\
 & & \mathcal{D}[\![expr]\!] \\
 & & Aux(3) := SaveSize(this) \\
 & & Aux(2) := ParAddr(this) \\
 & & Aux(1) := Aux(3) \oplus_i Aux(2) \\
 & & Aux(0) := \text{LDA } Aux(1) \\
 & & \mathcal{A}[\![Fields(Pri(this.1))]\!] \; Aux(0) \; Loc(Field(this.1)) \; FirstReg(this.1)
\end{array}$$

Makros:

$$\begin{array}{ll}
ProcLab(this) & \triangleq Lab(body(proc(prog, this))) \\
SaveSize(o) & \triangleq inttoquad(savesize(prog, o)) \\
ParAddr(o) & \triangleq inttoquad(paraddr(prog, o))
\end{array}$$

Funktionsaufruf

Transformation

$$\begin{array}{lll}
\mathcal{E}[\![call(\mathbf{id}, args)]\!] & = & \text{SAVE} \qquad \textbf{if } isAtomic(this) \\
 & & \text{PUSH } ParsSize(this) \\
 & & \mathcal{E}[\![args]\!] \\
 & & Val(this) : \text{CALL } ProcLab(this), SaveSize(this) \\
 & & \text{POP } ParsSize(this) \\
 & & \text{RESTORE} \\
 & & Aux(1) := O^{64} \\
 & & Val(this) := RetReg(this) \oplus Aux(1) \\
 & = & \text{SAVE} \qquad \textbf{if } isStruct(Pri(this)) \\
 & & \text{PUSH } ParsSize(this) \\
 & & \mathcal{E}[\![args]\!] \\
 & & Val(this) : \text{CALL } ProcLab(this), SaveSize(this) \\
 & & \text{RESTORE} \\
 & & \text{POP } ParsSize(this) \\
 & & \mathcal{A}[\![Fields(Pri(this))]\!] \; Val(this) \; RetReg(this) \; FirstReg(this)
\end{array}$$

Makros: $RetReg(this) \triangleq mkreg(retreg(prog, this))$

Lemma 4.13 (Korrektheit des Prozeduraufrufs und Funktionsaufrufs)

Mit ρ, dem Prozedur– bzw. Funktionsaufruf *call* und der Transformation $\mathcal{S}[\![\cdot]\!]$ bzw. $\mathcal{E}[\![\cdot]\!]$ wird eine *n*-*m*-Simulation definiert, sofern die Bedingungen der statischen Semantik erfüllt sind.

Beweisidee

Direkte Simulation bis zum CALL-Befehl

Prozedur– und Funktionsrückkehr

Transformation:

$$\begin{array}{lll}
\mathcal{S}[\![return]\!] & = & \text{RET} \\
\mathcal{S}[\![return(expr)]\!] & = & \mathcal{E}[\![expr]\!] \qquad \textbf{if } isAtomic(this) \\
 & & Aux(1) := O^{64} \\
 & & Val(this) := Val(this.0) \oplus_l O^{64} \\
 & & \text{RET} \\
 & = & \mathcal{D}[\![expr]\!] \qquad \textbf{if } isStruct(Pri(this.0)) \\
 & & Aux(1) := O^{64} \\
 & & Val(this) := Loc(this.0) \oplus_l O^{64} \\
 & & \text{RET} \\
\mathcal{P}[\![procdecl(type, \mathbf{id}, pars, stat)]\!] & = & \mathcal{S}[\![stat]\!] \\
 & & \text{RET}
\end{array}$$

Lemma 4.13 (Korrektheit der Prozedur–bzw. und Funktionsrückkehr)

Mit ρ, dem Prozedur– bzw. Funktionsaufruf *call*, der Transformation $\mathcal{S}[\![\cdot]\!]$ bzw. $\mathcal{E}[\![\cdot]\!]$, der Transformation $\mathcal{P}[\![\cdot]\!]$ der aufgerufenen Prozedur bzw. Funktion sowie *return* und der Transformation $\mathcal{S}$ wird eine *n*-*m*-Simulation definiert, sofern die Bedingungen der statischen Semantik erfüllt sind.

Beweisidee

Direkte Simulation ab dem RET-Befehl und ab dem Befehl nach dem aufrufenden CALL-Befehl

Programme

Statische Semantik

Der Block des Programms p bekommt ein Label, d.h. es gilt $\mathfrak{D} \models bbBlock(first(p, 1)) \doteq true$ und das Programm bekommt ein Register zugeordnet, d.h. $\mathfrak{D} \models D(exprotoreg(p, ()))$ und $\mathfrak{D} \models numregs(p, ()) \doteq 1$

Transformation

$$\begin{array}{lll}
\mathcal{T}[\![prog(decls, block)]\!] & = & \text{PUSH SizeGlobals} \\
 & & Val(this : \text{CALL } Lab(first(prog, 1)) \\
 & & \mathcal{S}[\![block]\!] \\
 & & \mathcal{P}[\![decls]\!] \\
 & & \text{Korrekte Funktionen für Division und Rest} \\
\mathcal{P}[\![nodecls]\!] & = & \\
\mathcal{P}[\![decls_1(decls_2, decl)]\!] & = & \mathcal{P}[\![decls_2]\!] \\
 & & \mathcal{P}[\![decl]\!] \\
\mathcal{P}[\![vardecl]\!] & = & \\
\mathcal{P}[\![classdecl]\!] & = &
\end{array}$$

Lemma 4.14 (Korrektheit des Initialzustands)

Sei $\mathfrak{Z}$ der Zustand nach der Ausführung des CALL-Befehls ausgehend von einem Initialzustand $\mathfrak{I}_Z$ der IL-Semantik $\mathcal{Z}$. Dann gibt es einen Initialzustand $\mathfrak{I}$ der DEC-Alpha-Semantik $\mathcal{D}$ von C-- mit $(\mathfrak{Z}, \mathfrak{I}) \in \rho$

Beweisidee

Konstruktion von $\mathfrak{I}$ aus $\mathfrak{Z}$ gemäß ρ und zeigen, dass $\mathfrak{I}$ eine $\textsc{InitState}_\alpha$-Algebra ist.

Zusammenfassung

Anforderung 4.15 (Statische Funktionen)

Sei p ein C---Programm und $p' = \mathit{compile}(p)$ das in IL übersetzte Programm. Dann müssen folgende Eigenschaften erfüllt sein:

$$
\begin{array}{lll}
\mathfrak{D} & \models & \mathit{exprtoreg}(p,o) \leq_I \mathit{exprtoreg}(p,o') \wedge \mathit{exprtoreg}(p,o) > \mathit{exprtoreg}(p,o') - \mathit{numreg}(p,o') \\
 & & \quad \Rightarrow o = o' \\
\mathfrak{D} & \models & \mathit{bbBegin}(p, \mathit{next}(p,o)) \doteq \mathit{bbEnd}(p,o) \\
\mathfrak{D} & \models & \mathit{bbEnd}(p,o) \doteq \mathit{true} \Rightarrow \mathit{target}(p,o) \doteq \mathit{label}(p, \mathit{next}(p,o)) \\
\mathfrak{D} & \models & o \text{ **is** STATS} \Rightarrow \mathit{occ}(p,o.0) \doteq \mathit{mknostat} \vee \mathit{bbBegin}(p, \mathit{occ}(p,o.1)) \doteq \mathit{bbEnd}(p, \mathit{occ}(p,o.0.1)) \\
\mathfrak{D} & \models & \mathit{occ}(p,o) \textbf{ is } \text{IF} \Rightarrow \mathit{bbBegin}(p, \mathit{yes}(p,o)) \doteq \mathit{true} \\
\mathfrak{D} & \models & \mathit{occ}(p,o) \textbf{ is } \text{IF} \Rightarrow \mathit{bbBegin}(p, \mathit{no}(p,o)) \doteq \mathit{true} \\
\mathfrak{D} & \models & \mathit{occ}(p,o) \textbf{ is } \text{IF} \Rightarrow \mathit{bbBegin}(p, \mathit{next}(p,o)) \doteq \mathit{true} \\
\mathfrak{D} & \models & \mathit{occ}(p,o) \textbf{ is } \text{IF} \Rightarrow \mathit{bbEnd}(p,o.1) \doteq \mathit{true} \\
\mathfrak{D} & \models & \mathit{occ}(p,o) \textbf{ is } \text{IF} \Rightarrow \mathit{bbEnd}(p,o.2) \doteq \mathit{true} \\
\mathfrak{D} & \models & \mathit{occ}(p,o) \textbf{ is } \text{WHILE} \Rightarrow \mathit{bbEnd}(p,o.2) \doteq \mathit{bbBegin}(p, \mathit{yes}(p,o)) \\
\mathfrak{D} & \models & \mathit{occ}(p,o) \textbf{ is } \text{WHILE} \Rightarrow \mathit{bbBegin}(p, \mathit{no}(p,o)) \doteq \mathit{true} \\
\mathfrak{D} & \models & \mathit{occ}(p,o) \textbf{ is } \text{WHILE} \Rightarrow \mathit{bbBegin}(p, \mathit{first}(p,o)) \doteq \mathit{true} \\
\mathfrak{D} & \models & \mathit{occ}(p,o) \textbf{ is } \text{WHILE} \Rightarrow \mathit{bbEnd}(p,o) \doteq \mathit{true} \\
\mathfrak{D} & \models & \mathit{occ}(p,o) \textbf{ is } \text{BREAK} \Rightarrow \mathit{target}(p,o) \doteq \mathit{label}(\mathit{break}(p,o)) \\
\mathfrak{D} & \models & \mathit{occ}(p,o) \textbf{ is } \text{CONTINUE} \Rightarrow \mathit{target}(p,o) \doteq \mathit{label}(\mathit{continue}(p,o)) \\
\mathfrak{D} & \models & \mathit{occ}(p,o) \textbf{is } \text{PROCDECL} \Rightarrow \mathit{bbBegin}(p,o.3) \\
\mathfrak{D} & \models & \mathit{occ}(p,o) \textbf{is } \text{CALL} \\
 & & \Rightarrow \quad \mathit{savesize}(p, \mathit{arg}(p,o,i)) \doteq \mathit{length}(\mathit{exprproc}(pc)) * 16 + \mathit{FrameSize}(\mathit{reladdr}(p,o)) \\
\mathfrak{D} & \models & \mathit{occ}(p,o) \textbf{is } \text{CALL} \\
 & & \Rightarrow \quad \mathit{paraddr}(p, \mathit{arg}(p,o,i)) \doteq \mathit{addr}(\mathit{Proc}(p,o), \mathit{get}(\mathit{pars}(\mathit{Proc}(p,o),i))) \\
\mathfrak{D} & \models & \mathit{occ}(p,o) \textbf{is } \text{CALL} \\
 & & \Rightarrow \quad \mathit{occ}(p,o') \textbf{ is } \text{RETURN} \wedge \mathit{InBody}(\mathit{Proc}(p,o),o') \Rightarrow \mathit{retreg}(p,o) \doteq \mathit{exprtoreg}(p,o') \\
\mathfrak{Z} & \models & \mathit{getlab}(\mathit{getbef}(p',i)) \doteq \mathit{getlab}(\mathit{getbef}(p',j)) \wedge \neg \mathit{getlab}(\mathit{getbef}(p',i)) \doteq \mathit{mknolabel} \Rightarrow i \doteq j
\end{array}
$$

$$\llbracket \mathit{labtobef} \rrbracket_Z(p', \llbracket \mathit{mklab} \rrbracket_D(\llbracket \mathit{label}(p,o) \rrbracket_D)) = \llbracket \mathit{mappc}(p,o) \rrbracket_Z$$

Zusammenfassung

Satz 4.16 (Korrektheit der Übersetzung)

Sei $\mathcal{D}$ die DEC-Alpha Semantik für C--, $\mathcal{Z}$ die Semantik für die Zwischensprache *IL* und für die Zustände $\mathfrak{Z} \in \mathbb{Z}$, $\mathfrak{D} \in \mathbb{D}$ gelte: $(\mathfrak{Z}, \mathfrak{D}) \in \rho$ gdw.

$$
\begin{array}{ll}
\llbracket \mathit{mem}_\alpha \rrbracket_Z & = \llbracket \mathit{mem}_\alpha \rrbracket_D \\
\llbracket \mathit{inp}_\alpha \rrbracket_Z & = \llbracket \mathit{inp}_\alpha \rrbracket_D \\
\llbracket \mathit{out}_\alpha \rrbracket_Z & = \llbracket \mathit{out}_\alpha \rrbracket_D \\
\llbracket \mathit{fpcr} \rrbracket_Z & = \llbracket \mathit{fpcr} \rrbracket_D \\
\llbracket \mathit{UP} \rrbracket_Z & = \llbracket \mathit{UP} \rrbracket_D \\
\llbracket \mathit{BP} \rrbracket_Z & = \llbracket \mathit{BP} \rrbracket_D \\
\llbracket \mathit{SP} \rrbracket_Z & = \llbracket \mathit{SP} \rrbracket_D \\
\llbracket \mathit{HP} \rrbracket_Z & = \llbracket \mathit{HP} \rrbracket_D \\
\llbracket \mathit{prog}_i \rrbracket_Z & = \mathit{compile}(\llbracket \mathit{prog} \rrbracket_D) \\
\llbracket \mathit{val} \rrbracket_D(o) & = \llbracket \mathit{val} \rrbracket_Z(\llbracket \mathit{mkreg} \rrbracket_Z(\llbracket \mathit{exprtoreg}(\mathit{prog},o) \rrbracket_D) \\
\llbracket \mathit{loc} \rrbracket_D(o) & = \llbracket \mathit{val} \rrbracket_Z(\llbracket \mathit{mkreg} \rrbracket_Z(\llbracket \mathit{exprtoreg}(\mathit{prog},o) - 1 \rrbracket_D) \\
\llbracket \mathit{pc} \rrbracket_D & = \llbracket \mathit{first}(\mathit{occ}(\mathit{prog},\mathit{pc})) \rrbracket_Z \text{ für ein } o \text{ mit} \\
 & \qquad \llbracket \mathit{applytrafo}(\mathit{prog},o) \rrbracket_D \doteq \mathit{true}, \text{ oder} \\
\llbracket \mathit{pc} \rrbracket_D & = \llbracket \mathit{next}(\mathit{prog},\mathit{pc}) \rrbracket_Z \text{ für ein solches } o \\
\llbracket \mathit{mappc}(\mathit{prog},\mathit{pc}) \rrbracket_D & = \llbracket \mathit{ip} \rrbracket_Z
\end{array}
$$

Desweiteren sei Ω' die Signatur der beobachtbaren Zustände und $\alpha_Z : \mathrm{Alg}(\Sigma_Z) \to \mathrm{Alg}(\Omega')_Z$ mit $\alpha(\mathfrak{Z}) \triangleq \mathfrak{Z}|'_\Omega$ sowie $\alpha_D : \mathrm{Alg}(\Sigma_D) \to \mathrm{Alg}(\Omega')_Z$ mit $\alpha(\mathfrak{D}) \triangleq \mathfrak{D}|'_\Omega$ die Abbildungen zu den beobachtbaren Zuständen. Falls die Bedingungen in Anforderung 4.15 erfüllt sind, ist ρ eine *n*-*m*-Simulation

Korollar 4.17 (Korrektheit der Übersetzung)

Für jedes C---Programm p gilt: $\mathcal{T}\llbracket p \rrbracket$ ist korrekte Übersetzung von p, sofern Anforderung 3.2 und die Anforderungen 4.15 erfüllt sind.

Beweis: Folgt aus Satz 4.16, Satz 3.22, Satz 1.3 und Satz 1.2

4.5 Übersetzungsvalidierung

Beobachtungen

- Für die Korrektheit müssen die Anforderungen 3.2 und 4.15 nachgewiesen werden
- Außerdem muss nachgewiesen werden, dass ein Übersetzer tatsächlich die Transformation $\mathcal{T}$ angewendet hat.

Lösungsmöglichkeiten

- Verifikation der Zwischencodegenerierung und der dazugehörigen Attributierung
- Programmprüfung

Fakt

Erste Alternative ist praktisch nicht durchführbar!

Programmprüfung

Beobachtungen

- Die Eigenschaften aus Anforderung 3.2 und bis auf die letzen beiden Eigenschaften in Anforderung 4.15 können alle an Hand des attributierten Strukturbaums überprüft werden.
- ☞ Im Wesentlichen müssen bestimmte Attribute, die nur der Übersetzer benötigt, auf Gleichheit geprüft werden.
- Die Eigenschaft $\mathfrak{Z} \models \mathit{getlab}(\mathit{getbef}(p',i)) \doteq \mathit{getlab}(\mathit{getbef}(p',j)) \wedge \neg \mathit{getlab}(\mathit{getbef}(p',i)) \doteq \mathit{mknolabel} \Rightarrow i \doteq j$ kann ausschließlich an Hand des übersetzten Programms überprüft werden
- Die Eigenschaft $\llbracket \mathit{labtobef} \rrbracket_Z(p', \llbracket \mathit{mklab} \rrbracket_D(\llbracket \mathit{label}(p,o) \rrbracket_D)) = \llbracket \mathit{mappc}(p,o) \rrbracket_Z$ kann unter Kenntnis des Quellprogramms p und des entsprechenden Zielprogramms p' überprüft werden
- ⇒ Implementierung eines verifizierten Programmprüfers, der als Eingaben das Quellprogramm p und dessen Übersetzung p' bekommt

Diskussion

Beobachtung

Mit Programmprüfung können die Voraussetzungen von Korollar 4.17 überprüft werden

Problem

Der Übersetzer könnte dem Programmprüfer einen falschen attributierten Strukturbaum mit einem falschen IL-Programm übergeben, damit der Programmprüfer die Übersetzung akzeptiert

Lösung

- Ein anderer verifizierter Programmprüfer überprüft, ob der attributierte Strukturbaum dem Quelltext entspricht
- Ein weiterer verifizierter Programmprüfer überprüft, ob das Zielprogramm tatsächlich eine Übersetzung des Quellprogramms ist

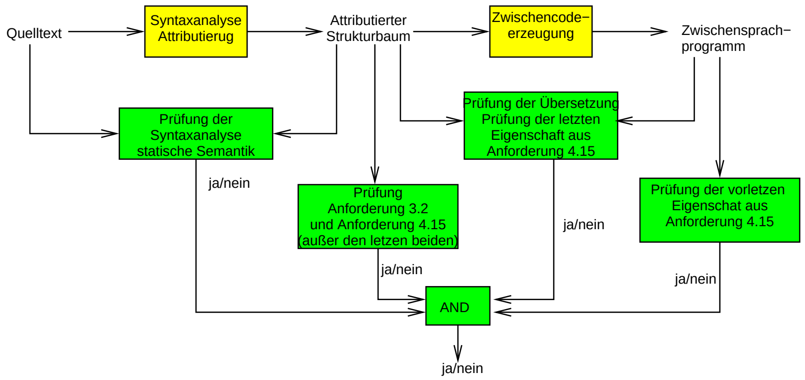
Überprüfung auf korrekte Übersetzung

- Annotation des attributierten Strukturbaums mit der angewendeten Regel
- Traversierung und Erzeugung des entsprechenden Codes

⇒ Nur wenn dieser Code identisch mit dem vom Compiler gelieferten Code ist, wird die Übersetzung akzeptiert

☞ Auch Vergleich der Menge von Grundblockgraphen auf Isomorphie genügt.

Übersetzungsvalidierung



- Die grünen Funktionen müssen verifiziert werden.

⇒ Zusammen mit Korollar 4.17 erhält man einen verifizierenden Übersetzer von `C--` nach *IL*, d.h. falls die Programmprüfung die Übersetzung akzeptiert, ist sie korrekt.