Kapitel 5
Korrektheit der Codeerzeugung und der Assemblierung

Wolf Zimmermann

Verifikation von Übersetzern

Inhalt

Ziele

- Kennenlernen der termersetzungsbasierten Codeerzeugung
- Verifikation der Codeerzeugung
- Bewusstsein für die Problematik des Zusammenwirkens von Registerzuteilung und Codeerzeugung
- Verifikation der Assemblierung

5.1 Einleitung

Aufgaben der Codeerzeugung

- Transformation des Zwischencodes nach Binärcode
- Ggf. Vorbereiten zum Binden bei getrennter Übersetzung

☞ Wird hier nicht betrachtet.

Vorgehen

Ausgangspunkt:

- Menge von Grundblockgraphen: Pro Prozedur und Hauptprogramm jeweilsein Grundblockgraph (Sicht BB)

☞ Zuordnung ist über Definitionstabelle bekannt

- Jeder Grundblock ist eine Folge von Bäumen (Sicht BB_T)
  - Zuweisung (ST-Befehle), SAVE, RESTORE, PUSH, POP, JMP, BEQ etc., CALL und RET sind Wurzeln von Bäumen
  - Die Bäumen sind über die Use-Def-Beziehungen definiert

Codeselektion: Für alle Nicht-Sprungbefehle wird der komplette Binärdcode selektiert, die Grundblockgraphenstrukturen bleiben erhalten

Assemblierung: Linearisierung der Menge der Grundblockgraphen und Auflösung der Sprünge

Methoden der Codeselektion

Makroexpansion

- Betrachtet jeden Zwischensprachbefehl als Folge von Maschinenbefehlen
- Register werden durch abstrakte Interpretation *on-the-fly* zugeteilt
- Ggf. wird *spill-code* erzeugt
- Einfach, aber liefert ineffizienten Code

Entscheidungstabellen

- Wie Makroexpansion, aber über Entscheidungstabellen werden Bedingungen für Alternativen definiert

☞ Besser als Makroexpansion, aber sehr aufwändig

Termersetzungssysteme

- Termersetzungssystem, das Bäume auf den Term • reduzieren
- Pro TES-Regel wird Code angegeben, der bei Anwendung erzeugt wird
- Ausnutzung algebraischer Identitäten ist möglich
- Registerzuteilung kann vorher geplant werden oder nachdem alle Regeln angewandt wurden erfolgen

☞ Wird hier betrachtet (mit Planung der Registerzuteilung vorher)

Diskussion

Beobachtungen

- Die Codeerzeugung verwendet die Sicht BB_T
- Die Optimierung verwendet häufig die Sicht BB

⇒ ASM-Semantik auf diesen Sichten

⇒ Korrektheitsbeweise für Sichtwechsel

Registersicherung und –wiederherstellung

- Reihenfolge der zu sichernden Register ist dem Übersetzer bekannt

☞ Analog spill-code

⇒ Übersetzer kann alten Zustand wiederherstellen

⇒ Registernummern müssen nicht mitgesichert werden

⇒ Optimierung

☞ Diese kann bereits auf *IL*-Ebene als korrekt nachgewiesen werden

Fazit

- Nachweis der Korrektheit der optimierten Zustandssicherung und -wiederherstellung
- Nachweis der Korrektheit des Sichtwechsels $IL \leftrightarrow BB$
- Nachweis der Korrektheit des Sichtwechsels $BB \Leftrightarrow BB_T$

Korrektheitsnachweise

Beweisidee zur Optimierung der Sicherung

- Definition des Laufzeitkellers der optimierten Version und optimierten analog Kapitel 3
- Vergleich der entsprechenden Aktivierungsvsrbunde auf dem Laufzeitkeller

☞ Identisch für den Speicher außer ungesicherten Variablen (wenn auch an unterschiedlichen Adressen)

☞ Vergleich der optimierten gesicherten Register mit den unoptimierten gesicherten Registern

Beweisidee für $IL \leftrightarrow BB$

Bijektive Zuordnung zwischen Befehlsnr. in Befehlssequenz und Befehlsnummer in einem Grundblock:

- Befehlszeiger im Grundblockgraph ist ein Paar aus Grundblocklabel und Befehlsnr.
- Mit $[\![ip]\!]_Z = [\![labtobef]\!]_Z(semprog_{iZ}, [\![bb(ip_B)]\!]_B) + [\![num(ip_B)]\!]_B$ wird eine 1-1-Simulation zwischen BB und IL definiert.

Korrektheitsnachweise

Beweisidee für Korrektheit $BB \leftrightarrow BB_T$

- Grundblockstruktur bleibt erhalten
- Definition einer abgeleiteten dynamischen Funktion *eval*, die Ausdrücke auswertet
- Semantik der Befehle werten Argument mit *eval* aus.
- Die Reihenfolge der Befehle ST, SAVE, RESTORE, PUSH, POP, JMP, BEQ etc., CALL und RET in einem Grundblock bleibt erhalten
- Jedes Register r in BB entspricht einem (Unter-)Baum $expr_r$ in BB_T (und umgekehrt)
- Für alle Register r mit nicht-lebendigen Werten an diesen Stellen in BB gilt $[\![val(r)]\!]_B = [\![eval(expr_r)]\!]_{BT}$
- Mit dieser Funktion wird eine n-1-Simulation definiert

Beispiel 5.1: Regel zum Speichern in BB_T

if CT **is** ST **then**
$Store(eval(Opd_1), eval(Opd_2))$
$Proceed$

$$\begin{array}{ll} eval(const(x)) & = x \\ eval(mkload(x)) & = Read(eval(x)) \\ eval(mkloada(x)) & = BP +_I eval(x) \\ eval(mkloadg(x)) & = UP +_I eval(x) \\ eval(x \oplus_I y) & = eval(x) +_I eval(y) \end{array}$$

5.2 Termersetzungssysteme

Beobachtungen (Weingart, 1973)

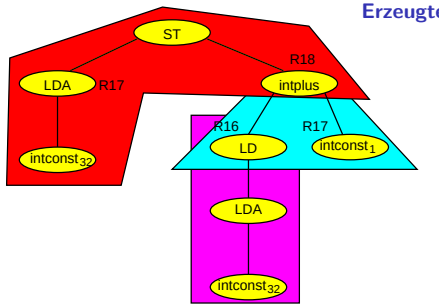
- Jeder aus Ausdrucksbaum e erzeugte Befehl b deckt Teil des Baums ab
- Gesamtcode überdeckt Gesamtbaum überlappungsfrei

Idee

- Jeder Ausdrucksbaum ist Term einer Termalgebra $\mathfrak{T}$
- Maschinenbefehle sind Terme einer anderen Termalgebra $\mathfrak{T}'$

⇒ Ersetze Ausdrucksbaum $e \in \mathfrak{T}$ durch Befehlssequenz $b \in \mathfrak{T}'$

Beispiel 5.2: Codeerzeugung mit TES



Erzeugter Code:

```
LDA R16,32(R30)
LDQ R16,R16
ADDQ R31,1,R17
ADDQ R16,R17,R16
LDA R17,32(R30)
STQ R16,R17
```

Regeln:

$intconst_j$	$\rightarrow reg_i$	$\{ADDQ\ R31, j, R_i\}$ falls $-128 \leq j \leq 127$
$intplus(reg_j, reg_k)$	$\rightarrow reg_i$	$\{ADDQ\ R_j, R_k, R_i\}$
LD reg_j	$\rightarrow reg_i$	$\{LDQ\ R_i, R_j\}$
LDA $intoconst_j$	$\rightarrow reg_i$	$\{LDA\ R_i, j(R30)\}$ falls $-2^{15} \leq j < 2^{15}$
ST reg_i reg_j	$\rightarrow \bullet$	$\{STQ\ R_i, R_j\}$

Weitere Regeln:

$intplus(reg_j, intconst_k)$	$\rightarrow reg_i$	$\{ADD\ R_j, k, R_i\}$falls $-2^{15} \leq k < 2^{15}$
LD (LDA $intconst_j$)	$\rightarrow reg_i$	$\{LDQ\ R_i, (k)R_j\}$ falls $-128 \leq j \leq 127$
ST (LDA $intconst_j$)	$\rightarrow reg_i$	$\{STQ\ R_i, (k)R_j\}$ falls $-128 \leq j \leq 127$

Erzeugter Code:

```
LDA R16(32),R30
ADDQ R16,1,R18
STQ R18,(32)R30
```


Diskussion

Beobachtungen

- Optimierungen durch bessere Regelabdeckungen
- ☞ Keinen Einfluss auf Korrektheit
- Registerzuteilung und Reihenfolge der Regelanwendungen muss mit dem Ermitteln von Regelabdeckungen geplant werden
- Für Korrektheit ist wichtig, dass kein Register zum Schreiben zugeteilt wird, das einen lebendigen Wert enthält
- Spill-Code kann mit integriert werden
- Werkzeuge erlauben Vollständigkeitsprüfung
- Trotz Notation können bereits binär codierte Befehle erzeugt werden
- Grundblockgraphstruktur bleibt erhalten
- ⇒ Grundblöcke enthalten bis auf Sprünge bereits binär codierte Maschinenbefehle (BB_α)
- Semantik von BB_α enthält jetzt nur noch die Register der DEC-Alpha und ersetzt die Zustandsübergangsregeln von BB_T durch die der DEC-Alpha. Bei Sprüngen wird auf Register statt auf Ausdrücke zugegriffen.

Fazit

Für die Korrektheit der Transformation $BB_T \rightarrow BB_\alpha$ muss nur die Korrektheit der TES-Regeln nachgewiesen und die Korrektheit der Annotationen geprüft werden.

5.3 Verifikation von termersetzungsbasierter Codeerzeugung

Beobachtung

- Grundblockstruktur bleibt erhalten
- ⇒ Bei Simulation muss nur gezeigt werden, dass der aus einer Regel erzeugte Code dem der Quelle entspricht
- ⇒ Befehlszeiger wird nur erhöht

Vorgehen

- Formalisierung der DEC-Alpha-Semantik
- Klassifikation der Simulationsbeweise
- Beispiele

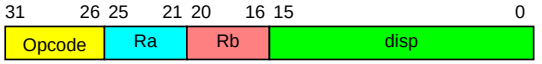
Formalisierung der DEC-Alpha-Semantik

Ausgangspunkt

Binärformat der Befehle

- Formalisierung der Zugriffsfunktionen und Befehlsklassifikation (als statische Funktionen)
- Makros zur Befehlszuordnung
- Formalisierung der Registertransfers

Binärformat für Speicherzugriffsbefehle



$opcode$: QUAD	$\rightarrow$ BITSEQ$_6$	$opcode(b)$	$\doteq sel_{32}^{31,26}(b)$
opd_1 : QUAD	$\rightarrow$ BITSEQ$_5$	$opd_1(b)$	$\doteq sel_{32}^{25,21}(b)$
opd_2 : QUAD	$\rightarrow$ BITSEQ$_5$	$opd_2(b)$	$\doteq sel_{32}^{20,16}(b)$
$disp$: QUAD	$\rightarrow$ BITSEQ$_{16}$	$disp(b)$	$\doteq sel_{32}^{15,0}(b)$

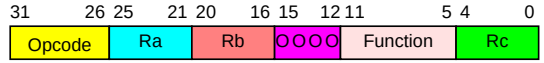
Für lange Sprünge wird dasselbe Format benutzt, nur dass *disp* für Hinweise zur Sprungvorhersage verwendet wird.

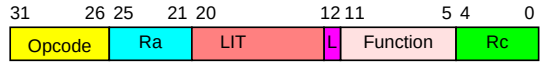
Formalisierung des Binärcodes

Binärformat für relativen Sprungbefehle

$jump$: QUAD $\rightarrow$ BITSEQ$_{21}$ $\quad jump(b) \doteq sel_{32}^{20,0}(b)$

Binärformat für arithmetische und logische Operationen





$dest$: QUAD	$\rightarrow$ BITSEQ$_5$	$dest(b)$	$\doteq sel_{32}^{4,0}(b)$
$constop$: QUAD	$\rightarrow$ BIT	$constop(b)$	$\doteq bit_{12}(b)$
lit : QUAD	$\rightarrow$ BITSEQ$_8$	$lit(b)$	$\doteq sel_{32}^{20,13}(b)$
$fnctn$: QUAD	$\rightarrow$ BITSEQ$_6$	$fnctn(b)$	$\doteq sel_{32}^{11,5}(b)$

Zustandsraum der DEC Alpha

Zustandsraum DEC-Alpha

spec STATE_α **extends** VALUE$_\alpha$
sorts BYTELIST
operations

mem_α :	QUAD	$\rightarrow$ QUAD
inp_α :		$\rightarrow$ BYTELIST
out_α :		$\rightarrow$ BYTELIST
$fpcr$:		$\rightarrow$ QUAD
reg :	BITSEQ$_5$	$\rightarrow$ QUAD
$freg$:	BITSEQ$_5$	$\rightarrow$ QUAD
bz :		$\rightarrow$ QUAD

axioms $reg(\mathrm{LLLLL}) \doteq \mathrm{O}^{64}$
$freg(\mathrm{LLLLL}) \doteq \mathrm{O}^{64}$

Zustandsraum BB_α

spec $\mathrm{STATE}_\alpha^{BB}$ **extends** VALUE$_\alpha$
sorts BYTELIST
operations

mem_α :	QUAD	$\rightarrow$ QUAD
inp_α :		$\rightarrow$ BYTELIST
out_α :		$\rightarrow$ BYTELIST
$fpcr$:		$\rightarrow$ QUAD
reg :	BITSEQ$_5$	$\rightarrow$ QUAD
$freg$:	BITSEQ$_5$	$\rightarrow$ QUAD
$prog_{BB_T}$:		$\rightarrow PROG_{BB_T}$
ip_{BB_T} :		$\rightarrow IP_{BB_T}$

axioms $reg(\mathrm{LLLLL}) \doteq \mathrm{O}^{64}$
$freg(\mathrm{LLLLL}) \doteq \mathrm{O}^{64}$

Makros zur Befehlserkennung

Grundprinzip

- Befehlszeiger bz verweist auf Befehl
- Klassifikation über *opcode* und ggf. *fnctn*

Beispiel 5.3: Klassifikation von Befehlen

Bef	$\triangle Read(bz)$
CI **is** LDQ	$\triangleq opcode(Bef) \doteq LOLOOL$
CI **is** STQ	$\triangleq opcode(Bef) \doteq$ LOLLOO
CI **is** BEQ	$\triangleq opcode(Bef) \doteq$ LLLOOL
CI **is** BR	$\triangleq opcode(Bef) \doteq$ LLOOOO
CI **is** ADDQ	$\triangleq opcode(Bef) \doteq$ OLOOOO $\wedge$ $fnctn(Bef) \doteq$ LOOOOOO $\wedge$ $bit_{12}(Bef) \doteq O$
CI **is** ADDQI	$\triangleq opcode(Bef) \doteq$ OLOOOO $\wedge$ $fnctn(Bef) \doteq$ LLOOOOO $\wedge$ $bit_{12}(Bef) \doteq L$
CI **is** SUBQ	$\triangleq opcode(Bef) \doteq$ OLOOOO $\wedge$ $fnctn(Bef) \doteq$ OLOLOOL $\wedge$ $bit_{12}(Bef) \doteq O$
CI **is** MULQ	$\triangleq opcode(Bef) \doteq$ OLOOLL $\wedge$ $fnctn(Bef) \doteq$ OLOOOOO $\wedge$ $bit_{12}(Bef) \doteq O$

Zustandsübergangsregeln

Grundprinzip

Formalisierung der Registertransfers der Maschinenspezifikation als ASM-Aktualisierungen

Beispiel 5.4: Zustandsübergangsregeln in DEC-Alpha

if CI **is** LDQ **then**
  if $\neg opd_1(Bef) \doteq$ LLLLL **then**
    $Opd_1 := Read(Opd_2 + sext_8(Disp))$
  $Proceed_\alpha$
if CI **is** LDA **then**
  if $\neg opd_1(Bef) \doteq$ LLLLL **then**
    $Opd_1 := Opd_2 + sext_8(Disp)$
  $Proceed_\alpha$
if CI **is** STQ **then**
  $Store(Opd_2, sext_8(Disp) +_I Opd_1)$
  $Proceed_\alpha$
if CI **is** ADDQ **then**
  if $\neg opd_3(Bef) \doteq$ LLLLL **then**
    $Opd_3 := Opd_1 +_I Opd_2$
  $Proceed_\alpha$
if CI **is** ADDQI **then**
  if $\neg opd_3(Bef) \doteq$ LLLLL **then**
    $Opd_3 := Opd_1 +_I sext_{16}(Lit)$
  $Proceed_\alpha$
if CI **is** BR **then**
  $Opd_1 := bz$
  $bz := bz +_I Sext_{32}(4 *_I Disp)$
if CI **is** BEQ **then**
  if $Opd_1 \doteq \mathrm{O}^{64}$ **then**
    $bz := bz +_I Sext_{32}(4 *_I Disp)$
  else $Proceed$

Makros:

Opd_i	$\triangleq reg(opd_i(Bef))$
$Disp$	$\triangleq disp(Read(Bef))$
Lit	$\triangleq lit(Read(Bef))$
$Proceed_\alpha$	$\triangleq bz := bz +_I 4$

Statische Funktion:

$sext_i$:	BITSEQ$_i$	$\rightarrow$ QUAD
$sext_i(\mathrm{O} : b)$	$\triangleq \mathrm{O}^{65-i}{+}{+}b$	
$sext_i(\mathrm{L} : b)$	$\triangleq \mathrm{L}^{65-i}{+}{+}b$	

Simulationsbeweise für Korrektheit von TES-Regeln

Statische Funktionen

regmap :	REGISTER	$\rightarrow$?BITSEQ$_5$	Registerzuteilung
fregmap :	REGISTER	$\rightarrow$?BITSEQ$_5$	Registerzuteilung
rule :	IP$_{BB_T}$	$\rightarrow$?RULE	angewandte Regel
sched :	IP$_{BB}$	$\rightarrow ?IP_{BB}$	nächste Stelle mit Regelanwendung

Anforderung 5.1 (Registerzuteilung)

- Es dürfen keine globalen Register zugeteilt werden (R14,R15,R29,R30)
- Register R31 und F31 dürfen nicht zugeteilt werden
- Register $regmap(lp_{BB_T})$ darf zum Zeitpunkt des Beschreibens keinen lebendigen Wert enthalten.

Definition der Simulationsrelation

Sei $\mathcal{B}_T$ die ASM für BB_T und $\mathcal{B}_\alpha$ die ASM für BB_α. Es ist $(\mathfrak{B}_\alpha, \mathfrak{B}_T) \in \rho$ gdw.:

- Speicher, Ein– und Ausgabestrom sowie Statusregister sind identisch
- Die lebendigen Inhalte der Register sowie der global verwendeten Register sind identisch mit den Werten der entsprechenden Teilausdrücke
- Die Befehlszeiger sind immer im Bildbereich von ρ. Sie sind immer Befehlszeigern im selben Grundblock zugeordnet, die den Anfang bzw. Ende der Übersetzung eines BB_T-Befehls betreffen.
- Dazu sei $size : \text{PROG}_{BB_T} \times \text{IP}_{BB_T} \rightarrow \text{INT}$ die Anzahl der bis zu einem Befehl generierten Befehle

$$
\begin{array}{lll}
[\![mem_\alpha]\!]_T & = [\![mem_\alpha]\!]_\alpha & (1)\\
[\![inp_\alpha]\!]_T & = [\![inp_\alpha]\!]_\alpha & (2)\\
[\![out_\alpha]\!]_T & = [\![out_\alpha]\!]_\alpha & (3)\\
[\![fpcr]\!]_T & = [\![fpcr]\!]_\alpha & (4)\\
[\![eval(t)]\!]_T & = [\![reg]\!]_\alpha([\![regmap(t)]\!]_B) \quad \text{falls } [\![isLeb]\!](eval(t), IP_T) & (5)\\
[\![BP]\!]_T & = [\![R30]\!]_\alpha & (6)\\
[\![SP]\!]_T & = [\![R29]\!]_\alpha & (7)\\
[\![HP]\!]_T & = [\![R15]\!]_\alpha & (8)\\
[\![UP]\!]_T & = [\![R14]\!]_\alpha & (9)\\
[\![eval(t)]\!]_T & = [\![freg]\!]_\alpha([\![fregmap(t)]\!]_B) \quad \text{falls } [\![isLeb]\!](eval(t), IP_T) & (10)\\
[\![ip]\!]_T & \in \text{RAN}(\rho) \quad \text{falls } \mathfrak{B}_T \models D(getbef_T(prog_T, ip) & (11)\\
[\![ip]\!]_\alpha & = \langle L, k\rangle \quad \text{falls } [\![ip]\!]_T = \langle L, i\rangle \text{ und } sembbsize_{B_T}(i) \leq k < [\![bbsize]\!]_{B_T}(i+1) & (12)
\end{array}
$$

Klassifikation der Simulationsbeweise

Beobachtung

Es gibt zwei Arten von Termersetzungsregeln:

- $t \rightarrow reg_i$ übersetzt einen echten Unterbaum. Dessen Wert muss in Register reg_i sein.
- $t \rightarrow \bullet$ übersetzt einen Befehl (z.B. ST) in einen entsprechenden Maschinenbefehl

Diskussion

- Bei der ersten Art von Regel wird nur *reg* oder *freg* verändert.
- Sei $\mathfrak{B}'$ Zustand nach dem Maschinencode. Dort muss gelten: $[\![eval]\!]_{\mathfrak{B}}(t) \doteq [\![reg]\!]_\alpha([\![regmap(t)]\!]_B)$ bzw. $[\![eval]\!]_{\mathfrak{B}}(t) \doteq [\![freg]\!]_\alpha([\![regmap(t)]\!]_B)$
- Bei der zweiten Art von Regel wird nur mem_α, die gloablen Register, das Statusregister oder die Ein-/Ausgabeströme verändert.
- Zu Beginn sind alle Operanden ausgewertet und die den Operanden zugeordneten Register enthalten deren Werte im Zustand vor den erzeugten Befehlen.

Korrektheit

Satz 5.2 (Korrektheit der TES-basierten Übersetzung)

Sei $\mathcal{B}_T$ und $\mathcal{B}_\alpha$ die ASMs, die die Semantik von BB_T bzw. BB_α definieren

- $\mathfrak{B}_T$ und $\mathfrak{B}_\alpha$ Zustände von $\mathcal{B}_T$ bzw. $\mathcal{B}_\alpha$, so dass $\langle \mathfrak{B}_\alpha, \mathfrak{B}_T\rangle \in \rho$
- $t \in \text{BEF}_T$ mit $[\![IP]\!]_B = t$
- t' ein echter Unterterm von t

Wenn die folgenden beiden Eigenschaften

i. Wenn $t'[reg_1, \ldots, reg_k] \rightarrow reg_i\{m_1; \ldots; m_n\}$ eine angewendete TES-Regel ist, $\mathfrak{B}'_\alpha$ Zustand mit $[\![lp]\!]_\alpha = m_1$ und $\langle \mathfrak{B}'_\alpha, \mathfrak{B}_T\rangle \in \rho$, dann ist auch $\langle \mathfrak{B}''_\alpha, \mathfrak{B}_T\rangle \in \rho$ für den Zustand $\mathfrak{B}''_\alpha$ nach Ausführung dieser Befehlssequenz.

ii. Für die TES-Regel $t[reg_1, \ldots, reg_k] \rightarrow \bullet\ \{m_1; \ldots; m_n\}$ wird ein Zustand $\mathfrak{B}'_\alpha$ mit $\mathfrak{B}'_\alpha \models ip_\alpha \doteq m_1$ und $\langle \mathfrak{B}'_\alpha, \mathfrak{B}_T\rangle \in \rho$ in einen Zustand $\mathfrak{B}''_\alpha$ mit $\langle \mathfrak{B}''_\alpha, \mathfrak{B}_T\rangle \in \rho$ nach Ausführung dieser Befehlssequenz überführt.

sowie Anforderung 5.1 erfüllt ist, dann definiert ρ eine n-m-Simulation

Beweisskizze

- Induktion über den Aufbau von BB_T-Befehlen beweist eine 1-n-Simulation
- Anwendung von Satz 1.3 (Vertikale Dekomposition)

Beispiel 5.5: Additionsbefehl

TES-Regel: $reg_1 \oplus_I reg_2 \rightarrow reg_3$ {ADDQ reg_1, reg_2, reg_3}

Nachweis der Eigenschaft Satz 5.2(i): Sei $\mathfrak{B}_T$ und $\mathfrak{B}_\alpha$, so dass $\langle \mathfrak{B}_\alpha, \mathfrak{B}_T \rangle \in \rho$.

$\Rightarrow$ Zu zeigen: $\langle \mathfrak{B}'_\alpha, \mathfrak{B}_T \rangle \in \rho$ für Nachfolgezustand $\mathfrak{B}'_\alpha$

- Sei $t = t_1 \oplus t_2$ der Baum, auf den die TES-Regel angewendet wird, $mapreg(t_1) = R_1$, $mapreg(t_2) = R_2$ und $mapreg(t) = R_3$

$\Rightarrow$
$$\begin{aligned}[\![eval(t)]\!]_{B_T} &= [\![eval(t_1 \oplus t_2)]\!]_{B_T} \\ &= [\![\oplus_I]\!]_{B_T}([\![eval(t_1)]\!]_{B_T}, [\![eval(t_2)]\!]_{B_T}) \\ &= [\![eval(t_1)]\!]_{B_T} +_I [\![eval(t_2)]\!]_{B_T} \\ &= [\![reg(R_1)]\!]_{B_\alpha} +_I [\![reg(R_2)]\!]_{B_\alpha}\end{aligned}$$

- Es gilt $Update(\mathcal{B}_A) = \{reg(R_3) := reg(R_1) +_I reg(R_2), Proceed_B\}$

$\Rightarrow$
$$\begin{aligned}[\![reg(R_3)]\!]_{B'_\alpha} &= [\![reg(R_1)]\!]_{B_\alpha} +_I [\![reg(R_2)]\!]_{B_\alpha} \\ &= [\![eval(t)]\!]_{B_T}\end{aligned}$$

$\Rightarrow$ $\langle \mathfrak{B}'_\alpha, \mathfrak{B}_T \rangle \in \rho$

Beispiel 5.6: Additionsbefehl mit kleiner Konstante

TES-Regel: $reg_1 \oplus_I intconst_j \rightarrow reg_3$ {ADDQI $reg_1, byte_0(j), reg_3$} falls $-128 \leq j < 127$

Nachweis der Eigenschaft Satz 5.2(i): Sei $\mathfrak{B}_T$ und $\mathfrak{B}_\alpha$, so dass $\langle \mathfrak{B}_\alpha, \mathfrak{B}_T \rangle \in \rho$.

$\Rightarrow$ Zu zeigen: $\langle \mathfrak{B}'_\alpha, \mathfrak{B}_T \rangle \in \rho$ für Nachfolgezustand $\mathfrak{B}'_\alpha$

- Sei $t = t_1 \oplus (\text{CONST } j)$ der Baum, auf den die TES-Regel angewendet wird, $mapreg(t_1) = R_1$ und $mapreg(t) = R_3$

$\Rightarrow$
$$\begin{aligned}[\![eval(t)]\!]_{B_T} &= [\![eval(t_1 \oplus (\text{CONST } j))]\!]_{B_T} \\ &= [\![\oplus_I]\!]_{B_T}([\![eval(t_1)]\!]B_T, [\![eval((\text{CONST } j))]\!]_{B_T}) \\ &= [\![eval(t_1)]\!]_{B_T} +_I j \\ &= [\![reg(R_1)]\!]_{B_\alpha} +_I j\end{aligned}$$

- Es gilt $Update(\mathcal{B}_A) = \{reg(R_3) := reg(R_1) +_I sext_8(byte_0(j)), Proceed_B\}$

$\Rightarrow$
$$\begin{aligned}[\![reg(R_3)]\!]_{B'_\alpha} &= [\![reg(R_1)]\!]_{B_\alpha} +_I [\![sext_8(byte_0(j))]\!]_\alpha \\ &= [\![eval(t)]\!]_{B_T}\end{aligned}$$

weil $\mathfrak{B}_\alpha \models -128 \leq j < 127 \Rightarrow j \doteq sext_8(byte_0(j))$

$\Rightarrow$ $\langle \mathfrak{B}'_\alpha, \mathfrak{B}_T \rangle \in \rho$

Laden einer kleinen Konstante

$intconst_j \rightarrow reg_1$ {ADDQI $R31, byte_0(j), reg_1$} falls $-128 \leq j < 127$

Beispiel 5.7: Lesen vom Speicher

TES-Regel: LD $reg_2 \rightarrow reg_1$ {LDQ $reg_1, 0(reg_2)$}

Nachweis der Eigenschaft Satz 5.2(i): Sei $\mathfrak{B}_T$ und $\mathfrak{B}_\alpha$, so dass $\langle \mathfrak{B}_\alpha, \mathfrak{B}_T \rangle \in \rho$.

$\Rightarrow$ Zu zeigen: $\langle \mathfrak{B}'_\alpha, \mathfrak{B}_T \rangle \in \rho$ für Nachfolgezustand $\mathfrak{B}'_\alpha$

- Sei $t = \text{LD } t_1$ der Baum, auf den die TES-Regel angewendet wird, $mapreg(t_1) = R_1$ und $mapreg(t) = R_2$

$\Rightarrow$
$$\begin{aligned}[\![eval(t)]\!]_{B_T} &= [\![eval(\text{LD } t_1)]\!]_{B_T} \\ &= [\![Read]\!]_{B_T}([\![eval(t_1)]\!]_{B_T}) \\ &= [\![Read]\!]_{B_\alpha}([\![reg(R_1)]\!]_{B_\alpha})\end{aligned}$$

- Es gilt $Update(\mathcal{B}_A) = \{reg(R_2) := Read(reg(R_1));\ Proceed_B\}$

$\Rightarrow$
$$\begin{aligned}[\![reg(R_2)]\!]_{B'_\alpha} &= [\![Read]\!]_{B_\alpha}([\![reg(R_1)]\!]_{B_\alpha}) \\ &= [\![eval(t)]\!]_{B_T}\end{aligned}$$

Beispiel 5.8: Lesen von kleinen Relativadressen

TES-Regel: LD (LDA (CONST j)) $\rightarrow reg_1$ {LDQ $reg_1, (\bar{j})(R30)$} falls $-2^{15} \leq j < 2^{15}$ und $\bar{j} \triangleq sel_{64}^{15,0}(j)$)

Nachweis der Eigenschaft Satz 5.2(i): Sei $\mathfrak{B}_T$ und $\mathfrak{B}_\alpha$, so dass $\langle \mathfrak{B}_\alpha, \mathfrak{B}_T \rangle \in \rho$.

$\Rightarrow$ Zu zeigen: $\langle \mathfrak{B}'_\alpha, \mathfrak{B}_T \rangle \in \rho$ für Nachfolgezustand $\mathfrak{B}'_\alpha$

- Sei $mapreg(t) = R_1$

$\Rightarrow$
$$\begin{aligned}[\![eval(t)]\!]_{B_T} &= [\![eval(\text{LD })]\!]_{B_T}([\![BP]\!]_{B_T} +_I j) \\ &= [\![Read]\!]_{B_T}([\![BP]\!]_{B_T} +_I j) \\ &= [\![Read]\!]_{B_\alpha}([\![reg(R30)]\!]_{B_\alpha} + j)\end{aligned}$$

- Es gilt

$Update(\mathcal{B}_A) = \{reg(R_2) := Read(reg(R30) +_I sext_{16}(sel_{64}^{15,0}(j)));\ Proceed_B\}$

$\Rightarrow$
$$\begin{aligned}[\![reg(R_1)]\!]_{B'_\alpha} &= [\![Read]\!]_{B_\alpha}([\![reg(R30)]\!]_{B_\alpha} +_I [\![sext_{16}(sel_{64}^{15,0}(j))]\!]_{B_\alpha}) \\ &= [\![eval(t)]\!]_{B_T}\end{aligned}$$

weil $\mathfrak{B}_\alpha \models -2^{15} \leq j < 2^{15} \Rightarrow j \doteq sext_16(sel_{64}^{15,0}(j))$

Beispiel 5.9: Laden von Relativadressen

TES-Regel: LDA (CONST j) $\rightarrow$ reg_1 {LDA $reg_1, (\bar{j})(R30)$} falls $-2^{15} \leq j < 2^{15}$ und $\bar{j} \triangleq sel_{64}^{15,0}(j)$)

Nachweis der Eigenschaft Satz 5.2(i): Sei $\mathfrak{B}_T$ und $\mathfrak{B}_\alpha$, so dass $\langle \mathfrak{B}_\alpha, \mathfrak{B}_T \rangle \in \rho$.

$\Rightarrow$ Zu zeigen: $\langle \mathfrak{B}'_\alpha, \mathfrak{B}_T \rangle \in \rho$ für Nachfolgezustand $\mathfrak{B}'_\alpha$

- Sei $t =$ LDA t_1 (CONST j) $mapreg(t_1) = R_1$ und $mapreg(t) = R_2$

$\Rightarrow$ $[\![eval(t)]\!]_{B_T} = [\![reg(R_1]\!]_{B_T} +_I j)$
$= [\![reg(R_1)]\!]_{B_\alpha} + j)$

- Es gilt

$Update(\mathcal{B}_A) = \{reg(R_2) := reg(R_1) +_I sext_{16}(sel_{64}^{15,0}(j))); Proceed_B\}$

$\Rightarrow$ $[\![reg(R_2)]\!]_{B'_\alpha} = [\![reg(R_1)]\!]_{B_\alpha} +_I [\![sext_{16}(sel_{64}^{15,0}(j))]\!]_{B_\alpha})$ weil
$= [\![eval(t)]\!]_{B_T}$

$\mathfrak{B}_\alpha \models -2^{15} \leq j < 2^{15} \Rightarrow j \doteq sext_16(sel_{64}^{15,0}(j))$

Laden einer mittelkleinen Konstante $intconst_j \rightarrow reg_1$ {LDA $reg_1, \bar{j}(R31)$} falls $-2^{15} \leq j < 2^{15}$

Beispiel 5.10: Laden einer mittleren Konstante

DEC-Alpha-Befehl LDAH: **if** CI **is** LDAH **then**
if $\neg opd_1 \doteq$ LLLLL **then**
$Opd_1 := Opd_2 + sext_{16}(Disp * 65536)$
$Proceed_\alpha$

TES-Regel: LDA (CONST j) $\rightarrow$ reg_1 {LDA $reg_1, J0(R31)$; LDAH $reg_1, J1(reg_1)$} falls $-2^{31} \leq j < 2^{31}$, $J0 \triangleq sel_{64}^{15,0}(j))$ und $J1 \triangleq sel_{64}^{31,16}(j))$

Nachweis der Eigenschaft Satz 5.2(i): Sei $\mathfrak{B}_T$ und $\mathfrak{B}_\alpha$, so dass $\langle \mathfrak{B}_\alpha, \mathfrak{B}_T \rangle \in \rho$.

$\Rightarrow$ Zu zeigen: $\langle \mathfrak{B}'_\alpha, \mathfrak{B}_T \rangle \in \rho$ für Nachfolgezustand $\mathfrak{B}'_\alpha$

- Sei $t =$ LDA t_1 (CONST j) $mapreg(t_1) = R_1$ und $mapreg(t) = R_2$

$\Rightarrow$ $semeval(t)_{B_T} = j$

- Es gilt

$Update(\mathcal{B}_A) = \{reg(R_2) := reg(R_1) +_I sext_{16}(sel_{64}^{15,0}(j))); Proceed_B\}$

$\Rightarrow$ $[\![reg(R_1)]\!]_{B''_\alpha} = [\![reg(R31)]\!]_{B_\alpha} +_I [\![sext_{16}(sel_{64}^{15,0}(j))]\!]_{B_\alpha}$
$= [\![sext_{16}(sel_{64}^{15,0}(j))]\!]_{B_\alpha}$

$\Rightarrow$ $Update(\mathcal{B}''_A) = \{reg(R_2) := reg(R_1) +_I sext_{16}(sel_{64}^{31,16}(j) *_I 65536)); Proceed_B\}$

$\Rightarrow$ $[\![reg(R_1)]\!]_{B'_\alpha} = [\![reg(R_1)]\!]_{B'_\rho} +_I [\![sext_{16}(sel_{64}^{31,16}(j) *_I 65536)]\!]_{B_\alpha}$
$= [\![sext_{16}(sel_{64}^{15,0}(j))]\!]_{B_\alpha} + [\![sext_{16}(sel_{64}^{31,16}(j) *_I 65536)]\!]_{B_\alpha}$

Problem

Falls $bit_{15}(j) =$ L ist $[\![sext_{16}(sel_{64}^{15,0}(j))]\!]_{B_\alpha} +_I [\![sext_{16}(sel_{64}^{31,16}(j) *_I 65536)]\!]_{B_\alpha} \neq j$

Beispiel 5.10: Abhilfe

Fehlerursache

- Vorzeichenerweiterung

$\Rightarrow$ Kein Befehl zum Unterdrücken der Vorzeichenerweiterung

if CI **is** ZAPI **then** Statische Funktion $bytezap$: QUAD $\times$
if $\neg opd_3 \doteq$ LLLLL **then**
$Opd_3 := bytezap(Opd_1, Disp)$
$Proceed_\alpha$

BYTE $\rightarrow$ QUAD mit $byte_i(bytezap(w, b)) \doteq O^8 \Leftrightarrow bit_i(b) \doteq$ L und $byte_i(bytezap(w, b)) \doteq byte_i(w) \Leftrightarrow bit_i(b) \doteq$ O

Korrektur

LDA (CONST j) $\rightarrow$ reg_1 { LDA $reg_1, J0(R31)$
ZAPreg_1, LLLLLLOO, reg_1
LDAH $reg_1, J1(reg_1)$}

Übung

Beweisen Sie nun die Korrektheit

Fazit

Wenn Anforderung 5.1 erfüllt ist, dann genügt es für jede Transformationsregel die entsprechende Simulation zu berechnen

- Im Prinzip handelt es sich um eine symbolische Auswertung der linken Seite und symbolische Zustandsübergänge auf der rechten Seite

$\Rightarrow$ Es entstehen einfache Gleichungen über Bitfolgen

$\Rightarrow$ Automatisierung möglich (realisiert in PVS)

5.4 Überprüfung der Korrektheit der Codegenerierung

Beobachtung

Anforderung 5.1 muss erfüllt sein, damit die Codeselektionsphase korrekt ist

- Mit den Attribute *regmap*, *fregmap* spezifiert der Übersetzer die Registerzuteilung, mit dem Attribut *rule* die anzuwendende Regel und mit dem Attribute *sched* wird eine Reihenfolge auf den anzuwendenden Regeln definiert

⇒ Wert in einem Register ist lebendig, wenn er mit einer Termersetzungsregel erzeugt wurde, aber noch nicht durch eine Termersetzungsregel verbraucht wurde

⇒ Kann statisch geprüft werden

⇒ Programmprüfung möglich

Überprüfung der Korrektheit der Anwendung der Transformation

- Überprüfung, ob annotierte Regel anwendbar
- Anwendung der Regeln gemäß *sched*
- Vergleich mit dem Resultat des Compilers

5.5 Assemblierung

Aufgabe

Abbildung der Grundblöcke in Speicher (mit Relativadressen) und Umsetzen der Sprungstruktur

Vorgehen

- Transformationsregeln für kurze, mittlere und große Sprünge.
- Der Übersetzer bildet die Sprungmarken auf Relativadressen ab: $labtoaddr : \text{LABEL} \to \text{QUAD}$
- Die Befehle in den Grundblöcken werden in derselben Reihenfolge im Speicher abgelegt.

⇒ Statische Funktion $bbsize : \text{LABEL} \to \text{QUAD}$

- Am Ende werden die Sprünge gemäß den obigen Transformationsregeln generiert

Beobachtungen

- Aufeinanderfolgende Grundblöcke die hintereinander angeordnet werden, dürfen sich nicht überlappen
- Kurze Sprünge können als Relativadresse in den Befehl codiert werden
- Mittlere und lange Sprünge erfordern das Laden des Sprungziels in ein Register

⇒ Optimierproblem

- Übersetzer berechnet pro Sprungziel in einem Grundblock, ob dieser Sprung eingespart werden kann, ein kurzer, mittelgroßer oder langer Sprung ist

⇒ Annotation mit Transformationsregeln

Assemblierung

Transformationsregeln

Kein Sprung: JMP $L \to \bullet\ \{\}$ falls
$labtoaddr(CB) +_I bbsize(CB) +_i 8 \doteq labtoaddr(L)$ und $CB \triangleq first(IP_{BB})$ der aktuelle Block ist.

Kurzer Sprung: JMP $L \to \bullet\ \{\text{BR } R1\ Disp\}$ falls
$labtoaddr(CB) +_I bbsize(CB) +_i 4 \doteq labtoaddr(L) +_I sext_{23}(disp *_I 4)$

Mittlerer Sprung: JMP $L \to \bullet\ \{$ BR $R1, O^{20}L$ falls
LDA $R2, sel_{64}^{15,0}(R31)$
ZAP $R2, \text{LLLLLLOO}, R2$
LDAH $R2, sel_{64}^{31,16}(R2)$
JMP $R1, R2\}$
$labtoaddr(CB) +_I bbsize(CB) +_i 4 \doteq labtoaddr(L) +_I sext_{32}(dist)$

Ganz langer Sprung: Muss dann noch zusätzlich Linksshifts einbauen

Korrektheit der Assemblierung

Relation ρ

Sei $\mathcal{B}_\alpha$ die ASM für die Semantik von BB_α und $\mathcal{A}$ die Semantik der Maschinensprache sowie $\mathfrak{B}_\alpha$ und $\mathfrak{A}$ zwei Zustände von $\mathcal{B}_\alpha$ bzw. $\mathcal{A}$. Es sei $\langle \mathfrak{B}_\alpha, \mathfrak{A} \rangle \in \rho$ gdw.

$[\![mem_\alpha]\!]_{B_\alpha}(x) = [\![mem_\alpha]\!]_{B_\alpha}(x)$ für $x \geq [\![reg(14)]\!]_{B_\alpha}$
$[\![reg]\!]_{B_\alpha} = [\![reg_\alpha]\!]_A$
$[\![freg]\!]_{B_\alpha} = [\![freg_\alpha]\!]_A$
$[\![inp_\alpha]\!]_{B_\alpha} = [\![inp_\alpha]\!]_A$
$[\![out_\alpha]\!]_{B_\alpha} = [\![out_\alpha]\!]_A$
$[\![bz]\!]_A = iptoaddr(ip)$
$\mathfrak{B}_\alpha \models D(getbef(prog_\alpha, x)) \Rightarrow iptoaddr(x) < reg(14)$
$[\![Read]\!]_A([\![iptoaddr(x)]\!]_{B_\alpha}) = [\![getbef(prog_\alpha, x)]\!]_{B_T}$

wobei $iptoaddr : \text{IP}_B \to \text{QUAD}$ statische Funktion mit $iptoaddr(\langle I, i \rangle) \doteq labtoaddr(I) + inttoquad(i * 4)$ ist.

Satz 5.3 (Korrektheit der Transformation für Sprünge)

ρ definiert eine 1-n-Simulation.

Korollar 5.4 (Korrektheit der Übersetzung)

Die Übersetzung von C-- zu DEC-Alpha-Assembler Code ist korrekt.

Beobachtung

Die letzen beiden Eigenschaften der Relation ρ können statisch überprüft werden.

Programmprüfung

Beweisverpflichtungen

- Die Grundblöcke dürfen sich im Speicher nicht überlappen
- Die anderen Befehle müssen im Speicher in derselben Reihenfolge wie im Grundblock vorkommen
- Die Sprünge müssen korrekt abgebildet sein

$$\mathfrak{B}_\alpha \models \quad LabAddr(l) \leq LabAddr(l') + BBsize(l') + Jmp(l) \wedge \\ LabAddr(l) \geq LabAddr(l') \Rightarrow l \doteq l'$$

$[\![getbef]\!]_{B_\alpha}([\![prog]\!]_{B_\alpha}, x\rangle) = [\![Read]\!]_\alpha([\![iptoaddr(x)]\!]_{B_\alpha})$

Sprünge als Übung

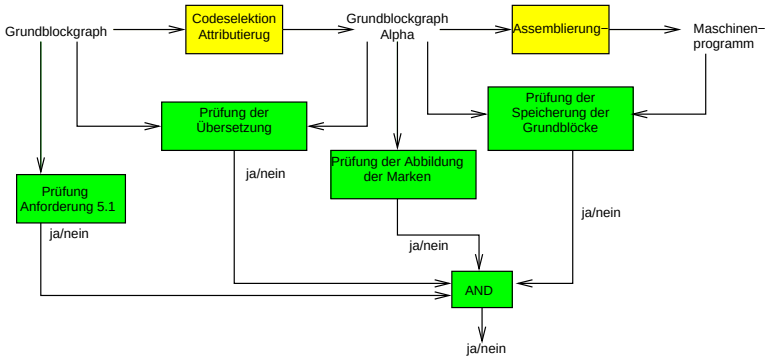
$LabAddr(l) \triangleq labeltoaddr(prog, l)$
$BBSize(l) \triangleq bbsize(prog, l)$
$Jmp(l) \triangleq jump(prog, l)$

Beobachtung

Diese Eigenschaften können statisch überprüft werden.

$\Rightarrow$ Programmprüfung stellt Korrektheit sicher

Übersetzungsvalidierung



- Die grünen Funktionen müssen verifiziert werden.

$\Rightarrow$ Zusammen mit Satz 5.4 ergibt sich ein verifizierendes Backend

$\Rightarrow$ Zusammen mit dem Resultat aus Kapitel 4 sich ein verifizierender Übersetzer