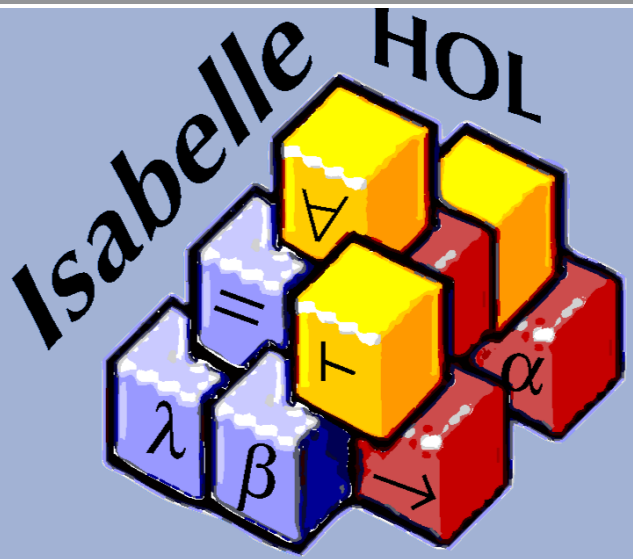
Karlsruher Institut für Technologie

Optimierungen in einer formalen Semantik
Konstantenfaltung und Konstantenpropagation

http://pp.info.uni-karlsruhe.de/lehre/SS2011/tba/

LEHRSTUHL PROGRAMMIERPARADIGMEN



Motivation

- Konstantenfaltung und -propagation sind wichtige Optimierungen in Compilern
- verringern *Registerdruck* (Anzahl der benötigten Register)
- Korrektheit dieser Optimierungen essentiell
- Korrektheit zu zeigen bzgl. formaler Semantik
- wir beschränken uns auf Konstantenfaltung und -propagation rein auf Semantikebene

Vor Erklärung der Algorithmen, Einführung in Maps, formale Semantik und Typsysteme

Karlsruher Institut für Technologie

Teil I

Maps

Map

Map: partielle Abbildung, also rechte Seite undefiniert für manche linke Seite

Typen inklusive Undefiniertheit in Isabelle: `'a option`

datatype `'a option = None | Some 'a`

- enthält alle Werte in `'a` mit `Some` vornangesetzt
- spezieller Wert `None` für Undefiniertheit

Beispiel: `bool option` besitzt die Elemente `None`, `Some True` und `Some False`

Map

also Maps in Isabelle von Typ `'a` nach `'b` haben Typ `'a ⇒ 'b option` oder kurz `'a ⇀ 'b` (⇀ ist `\<rightharpoonup>`)

- leere Map (also überall undefiniert): `empty`
- Update der Map `M`, so dass `x` auf `y` zeigt: `M(x ↦ y)` (↦: `|->`)
- Wert von `x` in Map `M` auf undefiniert setzen: `M(x := None)`
  (Hinweis: `M(x ↦ y)` entspricht `M(x := Some y)`)
- `x` hat in Map `M` Wert `y`, wenn gilt: `M x = Some y`
- `x` ist in Map `M` undefiniert, wenn gilt: `M x = None`
- um Map eigenen Typnamen zu geben: **type_synonym**
  Beispiel: **type_synonym** `nenv = nat ⇀ bool`

Falls mehr Infos zu Maps nötig: *Isabelle-Verzeichnis*/src/HOL/Map.thy

Karlsruher Institut für Technologie

Teil II

Formale Semantik

Was ist Semantik?

Zwei Konzepte bei Programmiersprachen (analog zu natürlicher Sprache), *Syntax* und *Semantik*

Was ist Semantik?

Zwei Konzepte bei Programmiersprachen (analog zu natürlicher Sprache), *Syntax* und *Semantik*

Syntax:

- Regeln für korrekte Anordnung von Sprachkonstrukten
- In Programmiersprachen meist durch Grammatik, vor allem in BNF (Backus-Naur-Form) gegeben
- Angegeben im Sprachstandard

Was ist Semantik?

Zwei Konzepte bei Programmiersprachen (analog zu natürlicher Sprache), *Syntax* und *Semantik*

Syntax:
- Regeln für korrekte Anordnung von Sprachkonstrukten
- In Programmiersprachen meist durch Grammatik, vor allem in BNF (Backus-Naur-Form) gegeben
- Angegeben im Sprachstandard

Semantik:
- Bedeutung der einzelnen Sprachkonstrukte
- Bei Programmiersprachen verschiedenste Darstellungsweisen:
  - informal (Beispiele, erläuternder Text etc.)
  - formal (Regelsysteme, Funktionen etc.)
- Angegeben im Sprachstandard (oft sehr vermischt mit Syntax)

Operationale Semantik

- Simuliert Zustandsübergänge auf abstrakter Maschine
- nahe an tatsächlichem Programmverhalten
- *Big-Step-Semantik*:
  Programm (= initiale Anweisung) + Startzustand wertet zu Endzustand aus
  Syntax: $\langle c, \sigma \rangle \Rightarrow \sigma'$ Anweisung c in Zustand σ liefert Endzustand σ'

einfache While-Sprache

arithmetische/boole'sche Ausdrücke: Zwei Werte `Intg` und `Bool`

- Konstanten `Val`
- Variablenzugriffe `Var`
- binäre Operatoren «`Eq`», «`And`», «`Less`», «`Add`»und «`Sub`»
  für ==, &&, $<$, +, -

einfache While-Sprache

arithmetische/boole'sche Ausdrücke: Zwei Werte `Intg` und `Bool`

- Konstanten `Val`
- Variablenzugriffe `Var`
- binäre Operatoren «`Eq`», «`And`», «`Less`», «`Add`»und «`Sub`»
  für ==, &&, $<$, +, -

Programmanweisungen:

- `Skip`
- Variablenzuweisung $x ::= e$
- sequentielle Komposition (Hintereinanderausführung) $c_1 \,;; c_2$
- if-then-else `IF` (b) c_1 `ELSE` c_2
- while-Schleifen `WHILE` (b) c'

einfache While-Sprache

arithmetische/boole'sche Ausdrücke: Zwei Werte `Intg` und `Bool`

- Konstanten `Val`
- Variablenzugriffe `Var`
- binäre Operatoren «`Eq`», «`And`», «`Less`», «`Add`»und «`Sub`»
  für ==, &&, $<$, +, -

Programmanweisungen:

- `Skip`
- Variablenzuweisung $x ::= e$
- sequentielle Komposition (Hintereinanderausführung) $c_1 ; ; c_2$
- if-then-else `IF` (b) c_1 `ELSE` c_2
- while-Schleifen `WHILE` (b) c'

Zustand:
beschreibt, welche Werte aktuell in den Variablen (Map)

Big Step Regeln

$[\![a]\!]\sigma$ Auswertung von arithm. oder boole'schem Ausdruck a in Zustand σ
Verwende Map, da Resultat undefiniert sein kann (z.B. bei 5 + `true`)

Big Step Regeln

$[\![a]\!]\sigma$ Auswertung von arithm. oder boole'schem Ausdruck *a* in Zustand σ
Verwende Map, da Resultat undefiniert sein kann (z.B. bei 5 + `true`)

$\Rightarrow$ *-Regeln* : $\langle \texttt{Skip}, \sigma \rangle \Rightarrow \sigma$

Big Step Regeln

$[\![a]\!]\sigma$ Auswertung von arithm. oder boole'schem Ausdruck a in Zustand σ
Verwende Map, da Resultat undefiniert sein kann (z.B. bei 5 + `true`)

$\Rightarrow$ *-Regeln* : $\quad \langle \texttt{Skip}, \sigma \rangle \Rightarrow \sigma \qquad \langle x ::= a, \sigma \rangle \Rightarrow \sigma(x \mapsto [\![a]\!]\sigma)$

Big Step Regeln

$[\![a]\!]\sigma$ Auswertung von arithm. oder boole'schem Ausdruck a in Zustand σ
Verwende Map, da Resultat undefiniert sein kann (z.B. bei 5 + `true`)

$\Rightarrow$ *-Regeln* : $\quad \langle \texttt{Skip}, \sigma \rangle \Rightarrow \sigma \qquad \langle x ::= a, \sigma \rangle \Rightarrow \sigma(x \mapsto [\![a]\!]\sigma)$

$$\frac{[\![b]\!]\sigma = \texttt{Some true} \qquad \langle c, \sigma \rangle \Rightarrow \sigma'}{\langle \texttt{IF } (b) \; c \; \texttt{ELSE } c', \sigma \rangle \Rightarrow \sigma'} \qquad \frac{[\![b]\!]\sigma = \texttt{Some false} \qquad \langle c', \sigma \rangle \Rightarrow \sigma'}{\langle \texttt{IF } (b) \; c \; \texttt{ELSE } c', \sigma \rangle \Rightarrow \sigma'}$$

Big Step Regeln

$[\![a]\!]\sigma$ Auswertung von arithm. oder boole'schem Ausdruck a in Zustand σ
Verwende Map, da Resultat undefiniert sein kann (z.B. bei 5 + `true`)

$\Rightarrow$ *-Regeln :* $\quad \langle \texttt{Skip}, \sigma \rangle \Rightarrow \sigma \qquad \langle x ::= a, \sigma \rangle \Rightarrow \sigma(x \mapsto [\![a]\!]\sigma)$

$$\frac{[\![b]\!]\sigma = \texttt{Some true} \qquad \langle c, \sigma \rangle \Rightarrow \sigma'}{\langle \texttt{IF } (b) \ c \ \texttt{ELSE } c', \sigma \rangle \Rightarrow \sigma'} \qquad \frac{[\![b]\!]\sigma = \texttt{Some false} \qquad \langle c', \sigma \rangle \Rightarrow \sigma'}{\langle \texttt{IF } (b) \ c \ \texttt{ELSE } c', \sigma \rangle \Rightarrow \sigma'}$$

$$\frac{[\![b]\!]\sigma = \texttt{Some true} \qquad \langle c', \sigma \rangle \Rightarrow \sigma' \qquad \langle \texttt{WHILE } (b) \ c', \sigma' \rangle \Rightarrow \sigma''}{\langle \texttt{WHILE } (b) \ c', \sigma \rangle \Rightarrow \sigma''}$$

$$\frac{[\![b]\!]\sigma = \texttt{Some false}}{\langle \texttt{WHILE } (b) \ c', \sigma \rangle \Rightarrow \sigma}$$

Big Step Regeln

$[\![a]\!]\sigma$ Auswertung von arithm. oder boole'schem Ausdruck a in Zustand σ
Verwende Map, da Resultat undefiniert sein kann (z.B. bei 5 + `true`)

$\Rightarrow$ *-Regeln* : $\quad \langle \texttt{Skip}, \sigma \rangle \Rightarrow \sigma \qquad \langle x ::= a, \sigma \rangle \Rightarrow \sigma(x \mapsto [\![a]\!]\sigma)$

$$\frac{[\![b]\!]\sigma = \texttt{Some true} \qquad \langle c, \sigma \rangle \Rightarrow \sigma'}{\langle \texttt{IF } (b) \ c \ \texttt{ELSE } c', \sigma \rangle \Rightarrow \sigma'} \qquad \frac{[\![b]\!]\sigma = \texttt{Some false} \qquad \langle c', \sigma \rangle \Rightarrow \sigma'}{\langle \texttt{IF } (b) \ c \ \texttt{ELSE } c', \sigma \rangle \Rightarrow \sigma'}$$

$$\frac{[\![b]\!]\sigma = \texttt{Some true} \qquad \langle c', \sigma \rangle \Rightarrow \sigma' \qquad \langle \texttt{WHILE } (b) \ c', \sigma' \rangle \Rightarrow \sigma''}{\langle \texttt{WHILE } (b) \ c', \sigma \rangle \Rightarrow \sigma''}$$

$$\frac{[\![b]\!]\sigma = \texttt{Some false}}{\langle \texttt{WHILE } (b) \ c', \sigma \rangle \Rightarrow \sigma} \qquad \frac{\langle c, \sigma \rangle \Rightarrow \sigma' \qquad \langle c', \sigma' \rangle \Rightarrow \sigma''}{\langle c; ; c', \sigma \rangle \Rightarrow \sigma''}$$

Ausführungsdauer

Erweiterung der Auswertungsfunktionen für Ausdrücke und der Big-Step-Semantik um einen Zeitbegriff:

Konstanten:	1
Variablen:	1
je bin. Operation:	1
Skip:	1
LAss:	1 + Auswertung des arith. Ausdrucks
If:	1 + Auswertung des bool. Ausdrucks + Dauer des gew. Zweigs
While-False:	1 + Auswertung des bool. Ausdrucks
While-True:	1 + Auswertung des bool. Ausdrucks + Dauer für Rumpf + Rest-Dauer

Formalisierung in Isabelle

Siehe Rahmen

Karlsruher Institut für Technologie

Teil III

Typsystem

Typsystem

Typsystem ordnet jedem Ausdruck Typ zu

Zwei Typen: `Boolean` und `Integer`

Typumgebung Γ: Map von Variablen nach Typ

Zwei Stufen:

1. jeder Ausdruck *e* bekommt unter Typumgebung Γ Typ *T* zugeordnet
   Syntax: $\Gamma \vdash e : T$
2. Anweisung *c* ist typbar unter Typumgebung Γ
   Syntax: $\Gamma \vdash c$

auch Typsystem definiert als induktive Menge

Regeln

Ausdrücke:

- Konstanten haben Typ des Werts
- Variablen haben den in Typumgebung gespeicherten Typ
- Operatoren haben, wenn Unterausdrücke Typen passend zu Operator, Typ des Resultats
  z.B. bei «*Less*»: Unterausdrücke *Integer*, ganzer Operator *Boolean*

Regeln

Ausdrücke:

- Konstanten haben Typ des Werts
- Variablen haben den in Typumgebung gespeicherten Typ
- Operatoren haben, wenn Unterausdrücke Typen passend zu Operator, Typ des Resultats
  z.B. bei «`Less`»: Unterausdrücke *`Integer`*, ganzer Operator *`Boolean`*

Anweisungen:

- `Skip` typt immer
- $x ::= e$ typt, wenn Typ der Variable x in Typumgebung Γ gleich Typ des Ausdruck e
- Sequenz typt, wenn beide Unteranweisungen typen
- `if` und `while` typen, wenn Unteranweisungen typen und Prädikat vom Typ *`Boolean`*

Formalisierung in Isabelle

Siehe Rahmen

Karlsruher Institut für Technologie

Teil IV

Algorithmen

Konstantenfaltung

Optimierung für Ausdrücke

- Wenn Berechnungen nur auf Konstanten, Ergebnis einfach einsetzen:
  `Val(Intg 5)` «`Add`» `Val(Intg 3)` wird zu `Val(Intg 8)`
  `Val(Intg 4)` «`Eq`» `Val(Intg 7)` wird zu `Val false`
- Wenn mind. eine Variable, einfach beibehalten:
  `Var y` «`Sub`» `Val(Intg 3)` bleibt `Var y` «`Sub`» `Val(Intg 3)`
- nicht sinvolle Ausdrücke auch beibehalten:
  `Val(Intg 5)` «`And`» `Val true` bleibt `Val(Intg 5)` «`And`» `Val true`
- Wenn Ausdruck nur Konstante oder Variable, auch beibehalten:
  `Val(Intg 5)` bleibt `Val(Intg 5)`, `Var y` bleibt `Var y`

Konstantenpropagation

Optimierung für Anweisungen

- Idee: Merken von Variablen, die konstant deklariert sind
- ermöglicht Ersetzen der Variable durch konstanten Wert
- dadurch möglich, `if`- Anweisungen zu vereinfachen
- Benötigt *Map* von Variablen nach Werten
- verwendet auch Konstantenfaltung

Beispiele

```
x ::= Val(Intg 7);;
y ::= Val(Intg 3);;
IF (Var x «Eq» Var y)
   (y ::= Var x «Add» Val(Intg 2))
ELSE (y ::= Var x «Sub» Var z);;
z ::= Var y
```

wird zu

```
x ::= Val(Intg 2);;
y ::= Var x;;
b ::= Var x «Eq» Var y;;
IF (Var b)
   (z ::= Var x «Add» Var y)
ELSE (z ::= Var x)
```

wird zu

Beispiele

```
x ::= Val(Intg 7);;
y ::= Val(Intg 3);;
IF (Var x «Eq» Var y)
   (y ::= Var x «Add» Val(Intg 2))
ELSE (y ::= Var x «Sub» Var z);;
z ::= Var y
```

wird zu

```
x ::= Val(Intg 7);;
y ::= Val(Intg 3);;
y ::= Val(Intg 7) «Sub» Var z;;
z ::= Var y
```

finale Map: $(x \mapsto \texttt{Val(Intg 7)})$

```
x ::= Val(Intg 2);;
y ::= Var x;;
b ::= Var x «Eq» Var y;;
IF (Var b)
   (z ::= Var x «Add» Var y)
ELSE (z ::= Var x)
```

wird zu

Beispiele

```
x ::= Val(Intg 7);;
y ::= Val(Intg 3);;
IF (Var x «Eq» Var y)
   (y ::= Var x «Add» Val(Intg 2))
ELSE (y ::= Var x «Sub» Var z);;
z ::= Var y
```

wird zu

```
x ::= Val(Intg 7);;
y ::= Val(Intg 3);;
y ::= Val(Intg 7) «Sub» Var z;;
z ::= Var y
```

finale Map: `(x ↦ Val(Intg 7))`

```
x ::= Val(Intg 2);;
y ::= Var x;;
b ::= Var x «Eq» Var y;;
IF (Var b)
   (z ::= Var x «Add» Var y)
ELSE (z ::= Var x)
```

wird zu

```
x ::= Val(Intg 2);;
y ::= Val(Intg 2);;
b ::= Val true;;
z ::= Val(Intg 4)
```

finale Map: `(x ↦ Val(Intg 2), y ↦ Val(Intg 2), b ↦ Val true, z ↦ Val(Intg 4))`

WHILE

Wie `IF` könnte man auch `WHILE` vereinfachen:

- falls Prädikat konstant *`false`*, komplettes `WHILE` durch `Skip` ersetzen
- falls Prädikat konstant *`true`*, Prädikat umschreiben, ansonsten Schleife beibehalten und in Schleifenkörper weiter Konstanten propagieren

WHILE

Wie IF könnte man auch WHILE vereinfachen:

- falls Prädikat konstant *false*, komplettes WHILE durch Skip ersetzen
- falls Prädikat konstant *true*, Prädikat umschreiben, ansonsten Schleife beibehalten und in Schleifenkörper weiter Konstanten propagieren

Problem: Konstanten im Schleifenkörper beeinflussen auch Prädikat! Beispiel:

```
x ::= Val(Intg 5);; y := Val(Intg 1);;
WHILE (Var x «Less» Val(Intg 7))
  (IF (Var y «Eq» Val(Intg 4))
    (x ::= Val(Intg 9))
   ELSE Skip;;
   y ::= Var y «Add» Val(Intg 1))
```

Darf das Prädikat von WHILE vereinfacht werden?

WHILE

- Kompletter Algorithmus bräuchte Fixpunktiteration!
- Zu kompliziert, deshalb Vereinfachung:
  Ist das Prädikat konstant `false` ist alles in Ordnung, ansonsten löschen wir beim `WHILE` die bisher gesammelte Konstanteninformation, verwenden also *`empty`* Map
- Ergebnis ist immer noch korrekt, aber nicht optimal vereinfacht
- Algorithmus so aber viel einfacher zu formalisieren

Projektaufgaben

1. Beweis, dass die vorgeg. Semantik deterministisch ist (sowohl im Endzustand, als auch im Zeitbegriff)
2. Formalisierung von Konstantenpropagation inklusive -faltung
3. Beweis, dass Konstantenpropagation Semantik erhält
   anders gesagt: “Endzustand ist der Gleiche, egal ob man im gleichen Anfangszustand Originalanweisung oder resultierende Anweisung der Konstantenpropagation verwendet”
4. Beweis, dass sich die Ausführungsgeschwindigkeit durch Konstantenpropagation erhöht
5. Beweis, dass Konstantenpropagation die Programmgröße verkleinert
6. Beweis, dass zwei-/mehrfache Anwendung der Konstantenpropagation das Programm nicht weiter verändert
7. Beweis, dass Konstantenpropagation Typisierung erhält
   anders gesagt: “Wenn Originalanweisung typt, dann auch resultierende Anweisung der Konstantenpropagation”

Beweise sollten verständlich und (komplett) in Isar geschrieben werden

Organisatorisches

- Projektbeginn: 31.05.2011
- Bearbeitung in Dreierteam
- Semantik und Typsystem als Isabelle-Rahmen vorgegeben
- Abgabe: 11.07.2011, 12.00 Uhr per Mail
- keine festen Dienstagstermine mehr
- stattdessen für Treffen Termine direkt mit mir ausmachen
- nach ca. 2 Wochen Formalisierung für Algorithmus vorlegen
- bei Problemen frühzeitig melden!

Hinweise

- Isabelle Dokumentation verwenden! Vor allem die Tutorials zu Isabelle/HOL, Isar und Function Definitions sollten helfen
- erst formalisieren, dann beweisen!
  Beispiele mittels *value* prüfen (z.B. Beispielprogramme in Semantics.thy)
- verwendet *quickcheck* und evtl. *nitpick* (`http://isabelle.in.tum.de/dist/Isabelle/doc/nitpick.pdf`) um Lemmas *vor* einem Beweis zu prüfen (spart oft unnötige Arbeit)
- falls Funktionsdefinitionen mit *fun* nicht funktionieren:
  - oftmals Probleme mit Termination
  - Fehlermeldung genau ansehen (wo Probleme mit Termination?)
    oft hilft eigene [simp] Regel
  - auch möglich: Zu *function* übergehen und versuchen, Termination explizit zu zeigen (siehe Tutorial zu Function Definitions)
- für die Beweise überlegen: welche Beziehungen müssen zwischen Semantikzustand, Typumgebung und Konstantenmap existieren?

Hinweise II

- `case`-Ausdrücke statt `if-then-else` verwenden wo möglich
  $\Rightarrow$ Entsprechende `split`-Regeln verwenden
  $\Rightarrow$ Mehr Automatismus

Beispiel

```
lemma "case v of None ⇒ f 0 | Some x ⇒ f x ⟹ ∃n. f n"
  by (cases v) auto
```

Hinweise II

- *case*-Ausdrücke statt *if-then-else* verwenden wo möglich
  $\Rightarrow$ Entsprechende *split*-Regeln verwenden
  $\Rightarrow$ Mehr Automatismus

Beispiel

```
lemma "case v of None ⇒ f 0 | Some x ⇒ f x ⟹ ∃n. f n"
  by (auto split: option.splits)
```