Theorembeweiser und ihre Anwendungen

Prof. Dr.-Ing. Gregor Snelting
Dipl.-Inf. Univ. Daniel Wasserrab

Lehrstuhl Programmierparadigmen
IPD Snelting
Universität Karlsruhe (TH)

Teil VII

Formale Verifikation eines C-Compilers

CompCERT

Projekt einer C-Compilerverifikation

- Aufteilung in

Frontend:	transformiert Sourcecode in Zwischensprache
Backend:	transformiert Zwischensprache in Assemblercode, kleinere Optimierungen

- Speichermodell in allen Sprachen und Zwischensprachen gleich
- Korrektheit: Semantik der Zielsprache macht "das Gleiche" wie Semantik der Ursprungssprache
- komplett verifiziert in Coq
- aus Coq-Beweisen OCaml-Code generierbar $\Rightarrow$ ausführbar

Projektseite: `http://compcert.inria.fr/`

X. Leroy and S. Blazy.
Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations.
Journal of Automated Reasoning, 41(1):1–31, Springer, 2008.
http://dx.doi.org/10.1007/s10817-008-9099-0

abstraktes Speichermodell und Werte

- konstant in allen Sprachen und Zwischensprachen
- Speicher *mem* modelliert als Menge von Blöcken
- Blöcke modelliert als Array von Bytes
- Speicherstelle *loc = (b,i)*, Block *b* mit Offset *i*
- jeder Block *b* in Speicher *M* untere Grenze $\mathcal{L}$*(M,b)* und obere Grenze $\mathcal{H}$*(M,b)*, Interval der gültigen Byte-Offsets von *b*
- initialer Speicherzustand *empty*

Werte der Semantiken:

val =	*int n*	— Integer-Werte
|	*float f*	— Fließkomma-Werte
|	*ptr loc*	— Pointer-Werte
|	*undef*	— undefinierter Wert

abstraktes Speichermodell und Werte

- konstant in allen Sprachen und Zwischensprachen
- Speicher *mem* modelliert als Menge von Blöcken
- Blöcke modelliert als Array von Bytes
- Speicherstelle *loc = (b,i)*, Block *b* mit Offset *i*
- jeder Block *b* in Speicher *M* untere Grenze $\mathcal{L}$*(M,b)* und obere Grenze $\mathcal{H}$*(M,b)*, Interval der gültigen Byte-Offsets von *b*
- initialer Speicherzustand *empty*

Werte der Semantiken:

```
val = int n       — Integer-Werte
    | float f     — Fließkomma-Werte
    | ptr loc     — Pointer-Werte
    | undef       — undefinierter Wert
```

Typen

werden später für *load* und *store* benötigt
Zweck:

- Größe und Anordnung ("alignment") der Daten darstellen
- Kompatibilität zwischen gespeicherten Daten und gelesenen Werten erzwingen, Art dynamischer Typcheck

```
memtype = float32 | float64        — Fließkommazahlen
    — "kleine" Integer
    | int8signed | int8unsigned | int16signed | int16unsigned
    | int32                         — Integer und Pointer
```

Operationen

Speichermodell stellt 4 grundlegende Operationen bereit:

- `alloc:: "(mem` $\times$ `int` $\times$ `int)` $\Rightarrow$ `(mem` $\times$ `block)"`
  `alloc(M,l,h) = (M',b)`
  Alloziert neuen Block mit Grenzen [`l`,`h`)
  gibt erweiterten Speicher `M'` und Referenz `b` auf neuen Block zurück
- `free:: "(mem` $\times$ `block)` $\Rightarrow$ `mem"`
  `free(M,b) = M'`
  Gibt Block `b` frei und angepassten Speicher `M'` zurück
- `load :: "(memtype` $\times$ `mem` $\times$ `block` $\times$ `int)` $\Rightarrow$ `val option"`
  `load(`τ`,M,b,n) = ⌊v⌋`
  Lese Bytes entsprechend τ in Block `b` mit Offset `n` von `M`
  falls erfolgreich, Inhalt der Bytes als Wert `v` zurückgegeben
- `store :: "(memtype` $\times$ `mem` $\times$ `block` $\times$ `int` $\times$ `val)` $\Rightarrow$ `mem option"`
  `store(`τ`,M,b,n,v) = ⌊M'⌋`
  Speichere `v` in Bytes entsprechend τ in Block `b` mit Offset `n` von `M`
  falls erfolgreich, aktualisierter Zustand `M'` zurückgegeben

Operationen

Speichermodell stellt 4 grundlegende Operationen bereit:

- `alloc:: "(mem` $\times$ `int` $\times$ `int)` $\Rightarrow$ `(mem` $\times$ `block)"`
  `alloc(M,l,h) = (M',b)`
  Alloziert neuen Block mit Grenzen [`l`,`h`)
  gibt erweiterten Speicher `M'` und Referenz `b` auf neuen Block zurück
- `free:: "(mem` $\times$ `block)` $\Rightarrow$ `mem"`
  `free(M,b) = M'`
  Gibt Block `b` frei und angepassten Speicher `M'` zurück
- `load :: "(memtype` $\times$ `mem` $\times$ `block` $\times$ `int)` $\Rightarrow$ `val option"`
  `load(`τ`,M,b,n) = ` $\lfloor v \rfloor$
  Lese Bytes entsprechend τ in Block `b` mit Offset `n` von `M`
  falls erfolgreich, Inhalt der Bytes als Wert `v` zurückgegeben
- `store :: "(memtype` $\times$ `mem` $\times$ `block` $\times$ `int` $\times$ `val)` $\Rightarrow$ `mem option"`
  `store(`τ`,M,b,n,v) = ` $\lfloor M' \rfloor$
  Speichere `v` in Bytes entsprechend τ in Block `b` mit Offset `n` von `M`
  falls erfolgreich, aktualisierter Zustand `M'` zurückgegeben

Operationen

Speichermodell stellt 4 grundlegende Operationen bereit:

- `alloc:: "(mem` $\times$ `int` $\times$ `int)` $\Rightarrow$ `(mem` $\times$ `block)"`
  `alloc(M,l,h) = (M',b)`
  Alloziert neuen Block mit Grenzen [*l*,*h*)
  gibt erweiterten Speicher *M'* und Referenz *b* auf neuen Block zurück
- `free:: "(mem` $\times$ `block)` $\Rightarrow$ `mem"`
  `free(M,b) = M'`
  Gibt Block *b* frei und angepassten Speicher *M'* zurück
- `load :: "(memtype` $\times$ `mem` $\times$ `block` $\times$ `int)` $\Rightarrow$ `val option"`
  `load(`τ`,M,b,n) = `$\lfloor$`v`$\rfloor$
  Lese Bytes entsprechend τ in Block *b* mit Offset *n* von *M*
  falls erfolgreich, Inhalt der Bytes als Wert *v* zurückgegeben
- `store :: "(memtype` $\times$ `mem` $\times$ `block` $\times$ `int` $\times$ `val)` $\Rightarrow$ `mem option"`
  `store(`τ`,M,b,n,v) = `$\lfloor$`M'`$\rfloor$
  Speichere *v* in Bytes entsprechend τ in Block *b* mit Offset *n* von *M*
  falls erfolgreich, aktualisierter Zustand *M'* zurückgegeben

Operationen

Speichermodell stellt 4 grundlegende Operationen bereit:

- `alloc:: "(mem` $\times$ `int` $\times$ `int)` $\Rightarrow$ `(mem` $\times$ `block)"`
  `alloc(M,l,h) = (M',b)`
  Alloziert neuen Block mit Grenzen [`l`,`h`)
  gibt erweiterten Speicher `M'` und Referenz `b` auf neuen Block zurück
- `free:: "(mem` $\times$ `block)` $\Rightarrow$ `mem"`
  `free(M,b) = M'`
  Gibt Block `b` frei und angepassten Speicher `M'` zurück
- `load :: "(memtype` $\times$ `mem` $\times$ `block` $\times$ `int)` $\Rightarrow$ `val option"`
  `load(`τ`,M,b,n) = `$\lfloor$`v`$\rfloor$
  Lese Bytes entsprechend τ in Block `b` mit Offset `n` von `M`
  falls erfolgreich, Inhalt der Bytes als Wert `v` zurückgegeben
- `store :: "(memtype` $\times$ `mem` $\times$ `block` $\times$ `int` $\times$ `val)` $\Rightarrow$ `mem option"`
  `store(`τ`,M,b,n,v) = `$\lfloor$`M'`$\rfloor$
  Speichere `v` in Bytes entsprechend τ in Block `b` mit Offset `n` von `M`
  falls erfolgreich, aktualisierter Zustand `M'` zurückgegeben

undef und Axiome

load liefert *undef*, falls vorher geschriebener und nun gelesener Bereich

1. bezüglich der Typen der Operationen nicht zusammenpassen, oder
2. sich partiell überlappen

viele Axiome, hier Beispiele zu

- Verhalten untereinander:

$$\texttt{alloc(M,l,h)} = \texttt{(M',b)} \;\wedge\; \texttt{b'} \neq \texttt{b} \Longrightarrow \texttt{load(}\tau\texttt{,M',b',n)} = \texttt{load(}\tau\texttt{,M,b',n)}$$
$$\texttt{free(M,b)} = \texttt{M'} \;\wedge\; \texttt{b'} \neq \texttt{b} \Longrightarrow \texttt{load(}\tau\texttt{,M',b',n)} = \texttt{load(}\tau\texttt{,M,b',n)}$$

- Gültigkeit von Block b in Speicher M, $M \models b$:

$$\texttt{alloc(M,l,h)} = \texttt{(M',b)} \Longrightarrow \neg(\texttt{M} \models \texttt{b})$$
$$\texttt{store(}\tau\texttt{,M,b,n,v)} = \lfloor \texttt{M'} \rfloor \Longrightarrow \texttt{M'} \models \texttt{b'} \Leftrightarrow \texttt{M} \models \texttt{b'}$$
$$\texttt{M} \models \texttt{b} \Longrightarrow \exists \texttt{M'}.\ \texttt{free(M,b)} = \texttt{M'}$$

- Aussagen über Blockgrenzen:

$$\texttt{alloc(M,l,h)} = \texttt{(M',b)} \Longrightarrow \mathcal{L}\texttt{(M',b)} = \texttt{l} \wedge \mathcal{H}\texttt{(M',b)} = \texttt{h}$$
$$\texttt{free(M,b)} = \texttt{M'} \wedge \texttt{b'} \neq \texttt{b} \Longrightarrow \mathcal{L}\texttt{(M',b')} = \mathcal{L}\texttt{(M',b')} \wedge \mathcal{H}\texttt{(M',b')} = \mathcal{H}\texttt{(M',b')}$$

undef und Axiome

load liefert *undef*, falls vorher geschriebener und nun gelesener Bereich

1. bezüglich der Typen der Operationen nicht zusammenpassen, oder
2. sich partiell überlappen

viele Axiome, hier Beispiele zu

- Verhalten untereinander:

$$\texttt{alloc}(M,l,h) = (M',b) \wedge b' \neq b \Longrightarrow \texttt{load}(\tau,M',b',n) = \texttt{load}(\tau,M,b',n)$$
$$\texttt{free}(M,b) = M' \wedge b' \neq b \Longrightarrow \texttt{load}(\tau,M',b',n) = \texttt{load}(\tau,M,b',n)$$

- Gültigkeit von Block b in Speicher M, $M \models b$:

$$\texttt{alloc}(M,l,h) = (M',b) \Longrightarrow \neg(M \models b)$$
$$\texttt{store}(\tau,M,b,n,v) = \lfloor M' \rfloor \Longrightarrow M' \models b' \Leftrightarrow M \models b'$$
$$M \models b \Longrightarrow \exists M'.\ \texttt{free}(M,b) = M'$$

- Aussagen über Blockgrenzen:

$$\texttt{alloc}(M,l,h) = (M',b) \Longrightarrow \mathcal{L}(M',b) = l \wedge \mathcal{H}(M',b) = h$$
$$\texttt{free}(M,b) = M' \wedge b' \neq b \Longrightarrow \mathcal{L}(M',b') = \mathcal{L}(M',b') \wedge \mathcal{H}(M',b') = \mathcal{H}(M',b')$$

`undef` und Axiome

`load` liefert `undef`, falls vorher geschriebener und nun gelesener Bereich

1. bezüglich der Typen der Operationen nicht zusammenpassen, oder
2. sich partiell überlappen

viele Axiome, hier Beispiele zu

- Verhalten untereinander:

$$\texttt{alloc(M,l,h)} = \texttt{(M',b)} \wedge \texttt{b'} \neq \texttt{b} \Longrightarrow \texttt{load}(\tau\texttt{,M',b',n}) = \texttt{load}(\tau\texttt{,M,b',n})$$

$$\texttt{free(M,b)} = \texttt{M'} \wedge \texttt{b'} \neq \texttt{b} \Longrightarrow \texttt{load}(\tau\texttt{,M',b',n}) = \texttt{load}(\tau\texttt{,M,b',n})$$

- Gültigkeit von Block b in Speicher M, $M \models b$:

$$\texttt{alloc(M,l,h)} = \texttt{(M',b)} \Longrightarrow \neg(\texttt{M} \models \texttt{b})$$

$$\texttt{store}(\tau\texttt{,M,b,n,v}) = \lfloor \texttt{M'} \rfloor \Longrightarrow \texttt{M'} \models \texttt{b'} \Leftrightarrow \texttt{M} \models \texttt{b'}$$

$$\texttt{M} \models \texttt{b} \Longrightarrow \exists\, \texttt{M'}.\ \texttt{free(M,b)} = \texttt{M'}$$

- Aussagen über Blockgrenzen:

$$\texttt{alloc(M,l,h)} = \texttt{(M',b)} \Longrightarrow \mathcal{L}\texttt{(M',b)} = \texttt{l} \wedge \mathcal{H}\texttt{(M',b)} = \texttt{h}$$

$$\texttt{free(M,b)} = \texttt{M'} \wedge \texttt{b'} \neq \texttt{b} \Longrightarrow \mathcal{L}\texttt{(M',b')} = \mathcal{L}\texttt{(M',b')} \wedge \mathcal{H}\texttt{(M',b')} = \mathcal{H}\texttt{(M',b')}$$

`undef` und Axiome

`load` liefert `undef`, falls vorher geschriebener und nun gelesener Bereich

1. bezüglich der Typen der Operationen nicht zusammenpassen, oder
2. sich partiell überlappen

viele Axiome, hier Beispiele zu

- Verhalten untereinander:

$$\texttt{alloc}(M,l,h) = (M',b) \wedge b' \neq b \Longrightarrow \texttt{load}(\tau,M',b',n) = \texttt{load}(\tau,M,b',n)$$
$$\texttt{free}(M,b) = M' \wedge b' \neq b \Longrightarrow \texttt{load}(\tau,M',b',n) = \texttt{load}(\tau,M,b',n)$$

- Gültigkeit von Block b in Speicher M, $M \models b$:

$$\texttt{alloc}(M,l,h) = (M',b) \Longrightarrow \neg(M \models b)$$
$$\texttt{store}(\tau,M,b,n,v) = \lfloor M' \rfloor \Longrightarrow M' \models b' \Leftrightarrow M \models b'$$
$$M \models b \Longrightarrow \exists M'.\ \texttt{free}(M,b) = M'$$

- Aussagen über Blockgrenzen:

$$\texttt{alloc}(M,l,h) = (M',b) \Longrightarrow \mathcal{L}(M',b) = l \wedge \mathcal{H}(M',b) = h$$
$$\texttt{free}(M,b) = M' \wedge b' \neq b \Longrightarrow \mathcal{L}(M',b') = \mathcal{L}(M',b') \wedge \mathcal{H}(M',b') = \mathcal{H}(M',b')$$

gültige Zugriffe

In Speicherzustand M gültig, Typ τ in Block b mit Offset n zu schreiben

Definition: $M \models \tau \mathbin{@} b, n \;\equiv\; M \models b \;\wedge\; \mathcal{L}(M,b) \leq n \;\wedge\; n + |\tau| \leq \mathcal{H}(M,b)$

Axiom: $M \models \tau \mathbin{@} b, n \Longrightarrow \exists M'.\ store(\tau, M, b, n, v) = \lfloor M' \rfloor$

Einfache Folgerungen:

- $alloc(M,l,h) = (M',b) \;\wedge\; l \leq n \;\wedge\; n + |\tau| \leq h \Longrightarrow M' \models \tau \mathbin{@} b, n$
- $alloc(M,l,h) = (M',b) \;\wedge\; M \models \tau \mathbin{@} b', n \Longrightarrow M' \models \tau \mathbin{@} b', n$
- $store(\tau, M, b, n, v) = \lfloor M' \rfloor \Longrightarrow M' \models \tau \mathbin{@} b', n \Leftrightarrow M \models \tau \mathbin{@} b', n$
- $free(M,b) = M' \Longrightarrow M' \models \tau \mathbin{@} b', n \Leftrightarrow M \models \tau \mathbin{@} b', n$

gültige Zugriffe

In Speicherzustand M gültig, Typ τ in Block b mit Offset n zu schreiben
Definition: $M \models \tau \mathbin{@} b,n \;\equiv\; M \models b \;\wedge\; \mathcal{L}(M,b) \leq n \;\wedge\; n + |\tau| \leq \mathcal{H}(M,b)$

Axiom: $M \models \tau \mathbin{@} b,n \Longrightarrow \exists M'.\ store(\tau,M,b,n,v) = \lfloor M' \rfloor$

Einfache Folgerungen:

- $alloc(M,l,h) = (M',b) \;\wedge\; l \leq n \;\wedge\; n + |\tau| \leq h \Longrightarrow M' \models \tau \mathbin{@} b,n$
- $alloc(M,l,h) = (M',b) \;\wedge\; M \models \tau \mathbin{@} b',n \Longrightarrow M' \models \tau \mathbin{@} b',n$
- $store(\tau,M,b,n,v) = \lfloor M' \rfloor \Longrightarrow M' \models \tau \mathbin{@} b',n \Leftrightarrow M \models \tau \mathbin{@} b',n$
- $free(M,b) = M' \Longrightarrow M' \models \tau \mathbin{@} b',n \Leftrightarrow M \models \tau \mathbin{@} b',n$

gültige Zugriffe

In Speicherzustand M gültig, Typ τ in Block b mit Offset n zu schreiben
Definition: $M \models \tau \,@\, b,n \;\equiv\; M \models b \;\wedge\; \mathcal{L}(M,b) \leq n \;\wedge\; n + |\tau| \leq \mathcal{H}(M,b)$

Axiom: $M \models \tau \,@\, b,n \implies \exists M'.\ store(\tau,M,b,n,v) = \lfloor M' \rfloor$

Einfache Folgerungen:

- $alloc(M,l,h) = (M',b) \;\wedge\; l \leq n \;\wedge\; n + |\tau| \leq h \implies M' \models \tau \,@\, b,n$
- $alloc(M,l,h) = (M',b) \;\wedge\; M \models \tau \,@\, b',n \implies M' \models \tau \,@\, b',n$
- $store(\tau,M,b,n,v) = \lfloor M' \rfloor \implies M' \models \tau \,@\, b',n \Leftrightarrow M \models \tau \,@\, b',n$
- $free(M,b) = M' \implies M' \models \tau \,@\, b',n \Leftrightarrow M \models \tau \,@\, b',n$

konkretes Speichermodell

Blöcke dargestellt als natürliche Zahlen: $block = nat$

Speicher mem dargestellt durch 4-Tupel (N,B,F,C) mit

- $N :: block$: erster, bisher nicht allozierter Block
- $B :: block \Rightarrow int \times int$: bestimmt Grenzen jeder Blockreferenz
- $F :: block \Rightarrow bool$: gibt an, ob Block bereits dealloziert oder nicht
- $C :: block \Rightarrow int \Rightarrow (memtype \times val)\ option$:
  weist Block b mit Offset n Inhalt zu mit

$None$	ungültig, oder
$Some\ (\tau, v)$	Wert v mit Typ τ

Realisierung der Operationen

- $\mathit{empty} = (0, \lambda b.\ [0,0), \lambda b.\ \mathit{false}, \lambda b.\ \lambda n.\ \mathit{None})$
- $\mathit{alloc}(M,l,h) = \mathit{let}\ b = N;$
  $M' = (N+1, B[b := [l,h)], F[b := \mathit{false}], C[b := \lambda n.\ \mathit{None}])\ \mathit{in}$
  $\mathit{if}\ \mathit{can\text{-}allocate}(M,h\text{-}l)\ \mathit{then}\ \mathit{Some}(b,M')\ \mathit{else}\ \mathit{None}$
- $\mathit{free}(M,b) = \mathit{if}\ \neg\ M \models b\ \mathit{then}\ \mathit{None}$
  $\mathit{else}\ \mathit{Some}(N, B[b := [0,0)], F[b := \mathit{true}], C)$
- $\mathit{store}(\tau,M,b,n,v) = \mathit{if}\ \neg\ M \models \tau\ @\ b,n\ \mathit{then}\ \mathit{None}$
  $\mathit{else}\ \mathit{let}\ c' = C(b)[n := \mathit{Some}(\tau,v), n+1 := \mathit{None}, \ldots,$
  $n + |\tau| - 1 := \mathit{None}]\ \mathit{in}\ \mathit{Some}(N,B,F,C[b := c'])$
- $\mathit{load}(\tau,M,b,n) = \mathit{if}\ \neg\ M \models \tau\ @\ b,n\ \mathit{then}\ \mathit{None}$
  $\mathit{else}\ \mathit{if}\ C(b)(n) = \mathit{Some}(\tau',v)\ \mathit{and}\ \langle \tau'\ \mathit{passt\ zu}\ \tau \rangle$
  $\mathit{and}\ C(b)(n+i) = \mathit{None}\ \mathit{for}\ i = 1,\ldots,|\tau| - 1$
  $\mathit{then}\ \mathit{Some}(v,\tau)\langle v\ \mathit{an}\ \tau\ \mathit{angepasst}\rangle\ \mathit{else}\ \mathit{Some}\ \mathit{undef}$

Realisierung der Operationen

- `empty = (0,` λ`b. [0,0),` λ`b. false,` λ`b.` λ`n. None)`
- `alloc(M,l,h) = let b = N;`
  `M' = (N + 1, B[b := [l,h)], F[b := false], C[b :=` λ`n. None]) in`
  `if can-allocate(M,h-l) then Some(b,M') else None`
- `free(M,b) = if` $\neg$ `M` $\models$ `b then None`
  `else Some(N, B[b := [0,0)], F[b := true], C)`
- `store(`τ`,M,b,n,v) = if` $\neg$ `M` $\models$ τ `@ b,n then None`
  `else let c' = C(b)[n := Some(`τ`,v), n + 1 := None, ... ,`
  `n + |`τ`| - 1 := None] in Some(N,B,F,C[b := c'])`
- `load(`τ`,M,b,n) = if` $\neg$ `M` $\models$ τ `@ b,n then None`
  `else if C(b)(n) = Some(`τ`',v) and` $\langle$`τ' passt zu τ`$\rangle$
  `and C(b)(n + i) = None for i = 1,...,|`τ`| - 1`
  `then Some(v,`τ`)`$\langle$`v an τ angepasst`$\rangle$ `else Some undef`

Realisierung der Operationen

- $\mathtt{empty} = (0, \lambda b.\ [0,0), \lambda b.\ \mathtt{false}, \lambda b.\ \lambda n.\ \mathtt{None})$
- $\mathtt{alloc}(M,l,h) = \mathtt{let}\ b = N;$
  $M' = (N+1, B[b := [l,h)], F[b := \mathtt{false}], C[b := \lambda n.\ \mathtt{None}])\ \mathtt{in}$
  $\mathtt{if}\ \mathtt{can\text{-}allocate}(M,h-l)\ \mathtt{then}\ \mathtt{Some}(b,M')\ \mathtt{else}\ \mathtt{None}$
- $\mathtt{free}(M,b) = \mathtt{if}\ \neg\ M \models b\ \mathtt{then}\ \mathtt{None}$
  $\mathtt{else}\ \mathtt{Some}(N, B[b := [0,0)], F[b := \mathtt{true}], C)$
- $\mathtt{store}(\tau,M,b,n,v) = \mathtt{if}\ \neg\ M \models \tau @ b,n\ \mathtt{then}\ \mathtt{None}$
  $\mathtt{else}\ \mathtt{let}\ c' = C(b)[n := \mathtt{Some}(\tau,v), n+1 := \mathtt{None}, \ldots,$
  $n + |\tau| - 1 := \mathtt{None}]\ \mathtt{in}\ \mathtt{Some}(N,B,F,C[b := c'])$
- $\mathtt{load}(\tau,M,b,n) = \mathtt{if}\ \neg\ M \models \tau @ b,n\ \mathtt{then}\ \mathtt{None}$
  $\mathtt{else}\ \mathtt{if}\ C(b)(n) = \mathtt{Some}(\tau',v)\ \mathtt{and}\ \langle \tau'\ \mathtt{passt\ zu}\ \tau \rangle$
  $\mathtt{and}\ C(b)(n+i) = \mathtt{None}\ \mathtt{for}\ i = 1,\ldots,|\tau| - 1$
  $\mathtt{then}\ \mathtt{Some}(v,\tau)\langle v\ \mathtt{an}\ \tau\ \mathtt{angepasst}\rangle\ \mathtt{else}\ \mathtt{Some\ undef}$

Realisierung der Operationen

- $\texttt{empty = (0, }\lambda\texttt{b. [0,0), }\lambda\texttt{b. false, }\lambda\texttt{b. }\lambda\texttt{n. None)}$
- $\texttt{alloc(M,l,h) = let b = N;}$
  $\texttt{M' = (N + 1, B[b := [l,h)], F[b := false], C[b := }\lambda\texttt{n. None]) in}$
  $\texttt{if can-allocate(M,h-l) then Some(b,M') else None}$
- $\texttt{free(M,b) = if }\neg\texttt{ M}\models\texttt{b then None}$
  $\texttt{else Some(N, B[b := [0,0)], F[b := true], C)}$
- $\texttt{store(}\tau\texttt{,M,b,n,v) = if }\neg\texttt{ M}\models\tau\texttt{ @ b,n then None}$
  $\texttt{else let c' = C(b)[n := Some(}\tau\texttt{,v), n + 1 := None, ...,}$
  $\texttt{n + |}\tau\texttt{| - 1 := None] in Some(N,B,F,C[b := c'])}$
- $\texttt{load(}\tau\texttt{,M,b,n) = if }\neg\texttt{ M}\models\tau\texttt{ @ b,n then None}$
  $\texttt{else if C(b)(n) = Some(}\tau\texttt{',v) and }\langle\tau\texttt{' passt zu }\tau\rangle$
  $\texttt{and C(b)(n + i) = None for i = 1,...,|}\tau\texttt{| - 1}$
  $\texttt{then Some(v,}\tau\texttt{)}\langle\texttt{v an }\tau\texttt{ angepasst}\rangle\texttt{ else Some undef}$

Realisierung der Operationen

- $\mathit{empty} = (0, \lambda b.\ [0,0), \lambda b.\ \mathit{false}, \lambda b.\ \lambda n.\ \mathit{None})$
- $\mathit{alloc}(M,l,h) = \mathit{let}\ b = N;$
  $M' = (N+1, B[b := [l,h)], F[b := \mathit{false}], C[b := \lambda n.\ \mathit{None}])\ \mathit{in}$
  $\mathit{if}\ \mathit{can\text{-}allocate}(M,h-l)\ \mathit{then}\ \mathit{Some}(b,M')\ \mathit{else}\ \mathit{None}$
- $\mathit{free}(M,b) = \mathit{if}\ \neg\ M \models b\ \mathit{then}\ \mathit{None}$
  $\mathit{else}\ \mathit{Some}(N, B[b := [0,0)], F[b := \mathit{true}], C)$
- $\mathit{store}(\tau,M,b,n,v) = \mathit{if}\ \neg\ M \models \tau\ @\ b,n\ \mathit{then}\ \mathit{None}$
  $\mathit{else}\ \mathit{let}\ c' = C(b)[n := \mathit{Some}(\tau,v), n+1 := \mathit{None}, \ldots,$
  $n + |\tau| - 1 := \mathit{None}]\ \mathit{in}\ \mathit{Some}(N,B,F,C[b := c'])$
- $\mathit{load}(\tau,M,b,n) = \mathit{if}\ \neg\ M \models \tau\ @\ b,n\ \mathit{then}\ \mathit{None}$
  $\mathit{else}\ \mathit{if}\ C(b)(n) = \mathit{Some}(\tau',v)\ \mathit{and}\ \langle \tau'\ \mathit{passt\ zu}\ \tau \rangle$
  $\mathit{and}\ C(b)(n+i) = \mathit{None}\ \mathit{for}\ i = 1,\ldots,|\tau| - 1$
  $\mathit{then}\ \mathit{Some}(v,\tau)\langle v\ \mathit{an}\ \tau\ \mathit{angepasst}\rangle\ \mathit{else}\ \mathit{Some}\ \mathit{undef}$

Realisierung der Operationen

- $\texttt{empty} = (0,\, \lambda b.\ [0,0),\, \lambda b.\ \texttt{false},\, \lambda b.\ \lambda n.\ \texttt{None})$
- $\texttt{alloc}(M,l,h) = \texttt{let}\ b = N;$
  $M' = (N+1,\, B[b := [l,h)],\, F[b := \texttt{false}],\, C[b := \lambda n.\ \texttt{None}])\ \texttt{in}$
  $\texttt{if can-allocate}(M,h-l)\ \texttt{then Some}(b,M')\ \texttt{else None}$
- $\texttt{free}(M,b) = \texttt{if}\ \neg\ M \models b\ \texttt{then None}$
  $\texttt{else Some}(N,\, B[b := [0,0)],\, F[b := \texttt{true}],\, C)$
- $\texttt{store}(\tau,M,b,n,v) = \texttt{if}\ \neg\ M \models \tau\ @\ b,n\ \texttt{then None}$
  $\texttt{else let}\ c' = C(b)[n := \texttt{Some}(\tau,v),\, n+1 := \texttt{None},\, \ldots,$
  $n + |\tau| - 1 := \texttt{None}]\ \texttt{in Some}(N,B,F,C[b := c'])$
- $\texttt{load}(\tau,M,b,n) = \texttt{if}\ \neg\ M \models \tau\ @\ b,n\ \texttt{then None}$
  $\texttt{else if}\ C(b)(n) = \texttt{Some}(\tau',v)\ \texttt{and}\ \langle \tau'\ \textit{passt zu}\ \tau \rangle$
  $\texttt{and}\ C(b)(n+i) = \texttt{None}\ \texttt{for}\ i = 1,\ldots,|\tau| - 1$
  $\texttt{then Some}(v,\tau)\langle v\ \textit{an}\ \tau\ \textit{angepasst}\rangle\ \texttt{else Some undef}$

einmal freigegebene Blöcke nie wieder alloziert $\Rightarrow$ unendlicher Speicher

Frontend

S. Blazy, Z. Dargaye, and X. Leroy.
Formal Verification of a C Compiler Front-End.
In *Proc. of Formal Methods*, volume 4085 of *LNCS*, pp. 460–475.
Springer, 2006.
http://dx.doi.org/10.1007/11813040_31

Quellsprache Clight

Clight Untermenge von C:

Typen: alle wesentlichen Typen von C inkl. Arrays, Pointer, function types
nicht enthalten: `struct`, `union`, `typedef`
Bitgrößen von Integern und Fließkomma spezifiziert (nicht in C!)

Ausdrücke: alle C-Operatoren (außer bezüglich `structs` und `unions`)
Seiteneffekte erlaubt, auch kombinierte Operatoren wie `x += y`
arithmetische Operatoren überladen

Anweisungen: alle strukturierten Kontrollanweisungen
(`if`, Schleifen, `break`, `continue`, `return`)
keine unstrukturierten (`goto`, `switch`, `longjmp`)

Variablen: globale und lokale `auto` Variablen erlaubt
Blockvariablen und `static` Variablen nur emulierbar

Clight Programm: Liste von Funktionsdefinitionen, Liste von Deklarationen globaler Variablen und Eintrittspunkt ins Programm

Quellsprache Clight

Clight Untermenge von C:

Typen: alle wesentlichen Typen von C inkl. Arrays, Pointer, function types
nicht enthalten: `struct`, `union`, `typedef`
Bitgrößen von Integern und Fließkomma spezifiziert (nicht in C!)

Ausdrücke: alle C-Operatoren (außer bezüglich `structs` und `unions`)
Seiteneffekte erlaubt, auch kombinierte Operatoren wie `x += y`
arithmetische Operatoren überladen

Anweisungen: alle strukturierten Kontrollanweisungen
(`if`, Schleifen, `break`, `continue`, `return`)
keine unstrukturierten (`goto`, `switch`, `longjmp`)

Variablen: globale und lokale `auto` Variablen erlaubt
Blockvariablen und `static` Variablen nur emulierbar

Clight Programm: Liste von Funktionsdefinitionen, Liste von Deklarationen globaler Variablen und Eintrittspunkt ins Programm

Quellsprache Clight

Clight Untermenge von C:

Typen: alle wesentlichen Typen von C inkl. Arrays, Pointer, function types
nicht enthalten: `struct`, `union`, `typedef`
Bitgrößen von Integern und Fließkomma spezifiziert (nicht in C!)

Ausdrücke: alle C-Operatoren (außer bezüglich `struct`s und `union`s)
Seiteneffekte erlaubt, auch kombinierte Operatoren wie `x += y`
arithmetische Operatoren überladen

Anweisungen: alle strukturierten Kontrollanweisungen
(`if`, Schleifen, `break`, `continue`, `return`)
keine unstrukturierten (`goto`, `switch`, `longjmp`)

Variablen: globale und lokale `auto` Variablen erlaubt
Blockvariablen und `static` Variablen nur emulierbar

Clight Programm: Liste von Funktionsdefinitionen, Liste von Deklarationen globaler Variablen und Eintrittspunkt ins Programm

Quellsprache Clight

Clight Untermenge von C:

Typen: alle wesentlichen Typen von C inkl. Arrays, Pointer, function types
nicht enthalten: `struct`, `union`, `typedef`
Bitgrößen von Integern und Fließkomma spezifiziert (nicht in C!)

Ausdrücke: alle C-Operatoren (außer bezüglich `structs` und `unions`)
Seiteneffekte erlaubt, auch kombinierte Operatoren wie `x += y`
arithmetische Operatoren überladen

Anweisungen: alle strukturierten Kontrollanweisungen
(`if`, Schleifen, `break`, `continue`, `return`)
keine unstrukturierten (`goto`, `switch`, `longjmp`)

Variablen: globale und lokale `auto` Variablen erlaubt
Blockvariablen und `static` Variablen nur emulierbar

Clight Programm: Liste von Funktionsdefinitionen, Liste von Deklarationen globaler Variablen und Eintrittspunkt ins Programm

Quellsprache Clight

Clight Untermenge von C:

Typen: alle wesentlichen Typen von C inkl. Arrays, Pointer, function types
nicht enthalten: `struct`, `union`, `typedef`
Bitgrößen von Integern und Fließkomma spezifiziert (nicht in C!)

Ausdrücke: alle C-Operatoren (außer bezüglich `structs` und `unions`)
Seiteneffekte erlaubt, auch kombinierte Operatoren wie `x += y`
arithmetische Operatoren überladen

Anweisungen: alle strukturierten Kontrollanweisungen
(`if`, Schleifen, `break`, `continue`, `return`)
keine unstrukturierten (`goto`, `switch`, `longjmp`)

Variablen: globale und lokale `auto` Variablen erlaubt
Blockvariablen und `static` Variablen nur emulierbar

Clight Programm: Liste von Funktionsdefinitionen, Liste von Deklarationen globaler Variablen und Eintrittspunkt ins Programm

Semantik von Clight

spezifiziert als Big-Step-Semantik
Auswertungsreihenfolge (im Gegensatz zu C) deterministisch

Semantik bestehend aus 7 Relationen:

$G, E \vdash a, M \stackrel{l}{\Rightarrow} loc, M'$	l-value Ausdrücke
$G, E \vdash a, M \Rightarrow v, M'$	r-value Ausdrücke
$G, E \vdash a^?, M \Rightarrow v, M'$	optionale Ausdrücke
$G, E \vdash a^*, M \Rightarrow v^*, M'$	Listen von Ausdrücken
$G, E \vdash s, M \Rightarrow out, M'$	Anweisungen, *out* Art der Termination
$G \vdash f(v^*), M \Rightarrow v, M'$	Funktionsaufrufe
$\vdash p \Rightarrow v$	Programme

wobei $v :: \texttt{val}$, $loc :: \texttt{loc}$, $M, M' :: \texttt{mem}$,
E lokale Umgebung: lokale Variablen nach Blockreferenzen,
G globale Umgebung: Variablen nach Blockreferenzen,
Referenzen nach Funktionsdefinitionen

Semantik von Clight

spezifiziert als Big-Step-Semantik
Auswertungsreihenfolge (im Gegensatz zu C) deterministisch

Semantik bestehend aus 7 Relationen:

$G, E \vdash a, M \stackrel{l}{\Rightarrow} loc, M'$	l-value Ausdrücke
$G, E \vdash a, M \Rightarrow v, M'$	r-value Ausdrücke
$G, E \vdash a^?, M \Rightarrow v, M'$	optionale Ausdrücke
$G, E \vdash a^*, M \Rightarrow v^*, M'$	Listen von Ausdrücken
$G, E \vdash s, M \Rightarrow out, M'$	Anweisungen, *out* Art der Termination
$G \vdash f(v^*), M \Rightarrow v, M'$	Funktionsaufrufe
$\vdash p \Rightarrow v$	Programme

wobei v :: `val`, loc :: `loc`, M, M' :: `mem`,
E lokale Umgebung: lokale Variablen nach Blockreferenzen,
G globale Umgebung: Variablen nach Blockreferenzen,
Referenzen nach Funktionsdefinitionen

Zielsprache Cminor

low-level imperative Sprache mit Ausdrücken, Anweisungen, Funktionen
Unterschiede zu Clight:

- arithmetische Operatoren nicht überladen, je nach Typ unterschiedlich
  explizites Casten nötig, keine kombinierten Operatoren wie `x += y`
- explizite Adressberechnungen (Adresse + Speicherplatz)
- nur 4 Anweisungen: `if`, `loop` (Endlosschleifen), `block` (Blöcke) und `exit` n (Verlassen des $n+1$ten umschließenden Blockes)
- in Funktionen, lokale Variablen nur für Skalare
  lokale Variablen nicht im Speicher $\Rightarrow$ keine Pointer darauf möglich
  stattdessen stack-allozierter Block, alloziert zu Funktionseintritt und automatisch freigegeben bei Verlassen, Zugriff mittels `addrstack`(n)

Semantik von Cminor

ähnlich wie Clight, aber lokale Umgebung durch Auswertung modifiziert

Big-Step, Semantikrelationen:

$G, sp, L \vdash a, E, M \rightarrow v, E', M'$	Ausdrücke
$G, sp, L \vdash a^*, E, M \rightarrow v^*, E', M'$	Listen von Ausdrücken
$G, sp \vdash s, E, M \rightarrow out, E', M'$	Anweisungen
$G \vdash fn(v^*), M \rightarrow v, M'$	Funktionsaufrufe
$\vdash p \rightarrow v$	Programme

E Abbildung lokale Variablen nach Werte (statt Adressen),
L Umgebung für `let`-gebundene Variablen in Ausdrücken,
sp Referenz auf Stack-Block der Funktion

Übersetzung

3 Aufgaben:

- typabhängige Operatoren einsetzen
  explizite Konversionen zwischen `ints` und `floats`
  für jeden Speicherzugriff explizit Speicherbereich angeben
- `while`, `do...while` und `for`-Schleifen in Endlosschleifen `loop` übersetzen mit Blöcken und Austrittspunkten
  `break` und `continue` werden zu passenden `exits`
- Clight Variablen ersetzen, entweder durch
  - lokale Variablen in Cminor
  - Teilbereiche des Cminor Stacks der aktuellen Funktion
  - global allozierte Speicherbereiche (globale Variablen)

realisiert als strukturell rekursive Coq Funktion
kann automatisch Caml-Code generieren, also Spezifikation ausführbar

Übersetzung

3 Aufgaben:

- typabhängige Operatoren einsetzen
  explizite Konversionen zwischen `ints` und `floats`
  für jeden Speicherzugriff explizit Speicherbereich angeben
- `while`, `do...while` und `for`-Schleifen in Endlosschleifen `loop`
  übersetzen mit Blöcken und Austrittspunkten
  `break` und `continue` werden zu passenden `exits`
- Clight Variablen ersetzen, entweder durch
  - lokale Variablen in Cminor
  - Teilbereiche des Cminor Stacks der aktuellen Funktion
  - global allozierte Speicherbereiche (globale Variablen)

realisiert als strukturell rekursive Coq Funktion
kann automatisch Caml-Code generieren, also Spezifikation ausführbar

Korrektheitsbeweis: Simulation

formale Semantiken sind Transitionssysteme,
bestehen aus Zuständen und Übergängen

Simulation:

- Standardtechnik für Äquivalenzbeweise zweier Transitionssysteme
- Idee: was man im einen Transitionssystem machen kann, kann man auch im anderen machen
- binäre Relation $\sim$ von Zuständen, beschreibt similare Zustände
- Zustände beider Transitionssysteme similar, nach einem (oder evtl. mehreren) Übergang wieder similar
- oftmals dazu Unterteilung: sichtbare bzw. unsichtbare Übergänge

Korrektheitsbeweis: Simulation

formale Semantiken sind Transitionssysteme,
bestehen aus Zuständen und Übergängen

Simulation:

- Standardtechnik für Äquivalenzbeweise zweier Transitionssysteme
- Idee: was man im einen Transitionssystem machen kann, kann man auch im anderen machen
- binäre Relation $\sim$ von Zuständen, beschreibt similare Zustände
- Zustände beider Transitionssysteme similar, nach einem (oder evtl. mehreren) Übergang wieder similar
- oftmals dazu Unterteilung: sichtbare bzw. unsichtbare Übergänge

Simulation zwischen Speicherzuständen

brauchen Funktion α, nimmt Clight Blockreferenz `b`, Rückgabe

- `None` falls Block kein Äquivalent in Cminor Speicherzustand
- `Some(b',`δ`)` falls `b` Subblock `b'` mit Offset δ in Cminor Speicherzustand

α definiert Relation zwischen Clight Werten und Cminor Werten:

$$\alpha \vdash \texttt{int } n \approx \texttt{int } n \qquad \alpha \vdash \texttt{float } n \approx \texttt{float } n \qquad \alpha \vdash \texttt{undef} \approx v$$

$$\frac{\alpha(b) = \texttt{Some}(b',\delta) \qquad n' = n + \delta \pmod{2^{32}}}{\alpha \vdash \texttt{ptr}(b,n) \approx \texttt{ptr}(b',n')}$$

Dann $\alpha \vdash M \approx M'$ Relation zwischen Clight und Cminor Speicherzuständen:

- $\alpha(b) = \texttt{Some}(b',\delta)$, v an b und v' an $(b',\delta) \Longrightarrow \alpha \vdash v \approx v'$
- $\alpha(b_1) = \texttt{Some}(b_1',\delta_1)$, $\alpha(b_2) = \texttt{Some}(b_2',\delta_2)$ und $b_1 \neq b_2$
  $\Longrightarrow b_1' \neq b_2'$ oder (b_1',δ_1) und (b_2',δ_2) nicht überlappend
- $\alpha(b) = \texttt{None}$ für alle bisher nicht allozierten Blöcke in M

Simulation zwischen Speicherzuständen

brauchen Funktion α, nimmt Clight Blockreferenz `b`, Rückgabe

- `None` falls Block kein Äquivalent in Cminor Speicherzustand
- `Some(b',`δ`)` falls `b` Subblock `b'` mit Offset δ in Cminor Speicherzustand

α definiert Relation zwischen Clight Werten und Cminor Werten:

$$\alpha \vdash \texttt{int n} \approx \texttt{int n} \qquad \alpha \vdash \texttt{float n} \approx \texttt{float n} \qquad \alpha \vdash \texttt{undef} \approx \texttt{v}$$

$$\frac{\alpha\texttt{(b) = Some(b',}\delta\texttt{)} \qquad \texttt{n' = n +}\ \delta\ \texttt{(mod}\ 2^{32}\texttt{)}}{\alpha \vdash \texttt{ptr(b,n)} \approx \texttt{ptr(b',n')}}$$

Dann $\alpha \vdash$ `M` $\approx$ `M'` Relation zwischen Clight und Cminor Speicherzuständen:

- α`(b) = Some(b',`δ`)`, `v` an `b` und `v'` an `(b',`δ`)` $\Longrightarrow \alpha \vdash$ `v` $\approx$ `v'`
- $\alpha(b_1)$ `=` $\mathit{Some}(b_1', \delta_1)$, $\alpha(b_2)$ `=` $\mathit{Some}(b_2', \delta_2)$ und $b_1 \neq b_2$
  $\Longrightarrow b_1' \neq b_2'$ oder (b_1', δ_1) und (b_2', δ_2) nicht überlappend
- α`(b) = None` für alle bisher nicht allozierten Blöcke in `M`

Simulation zwischen Speicherzuständen

brauchen Funktion α, nimmt Clight Blockreferenz b, Rückgabe

`None`	falls Block kein Äquivalent in Cminor Speicherzustand
`Some(b',`δ`)`	falls b Subblock b' mit Offset δ in Cminor Speicherzustand

α definiert Relation zwischen Clight Werten und Cminor Werten:

$$\alpha \vdash \texttt{int } n \approx \texttt{int } n \qquad \alpha \vdash \texttt{float } n \approx \texttt{float } n \qquad \alpha \vdash \texttt{undef} \approx v$$

$$\frac{\alpha(b) = \texttt{Some}(b',\delta) \qquad n' = n + \delta \pmod{2^{32}}}{\alpha \vdash \texttt{ptr}(b,n) \approx \texttt{ptr}(b',n')}$$

Dann $\alpha \vdash M \approx M'$ Relation zwischen Clight und Cminor Speicherzuständen:

- $\alpha(b) = \texttt{Some}(b',\delta)$, v an b und v' an $(b',\delta) \Longrightarrow \alpha \vdash v \approx v'$
- $\alpha(b_1) = \texttt{Some}(b_1',\delta_1)$, $\alpha(b_2) = \texttt{Some}(b_2',\delta_2)$ und $b_1 \neq b_2$
  $\Longrightarrow b_1' \neq b_2'$ oder (b_1',δ_1) und (b_2',δ_2) nicht überlappend
- $\alpha(b) = \texttt{None}$ für alle bisher nicht allozierten Blöcke in M

Korrektheitsbeweis durch Simulation

Simulationsaussagen der Semantiken: ($\mathcal{R}$,$\mathcal{L}$,$\mathcal{S}$ Übersetzungsfunktionen)
Wenn $\alpha \vdash M \approx M'$, dann gibt es Cminor Umgebung E_1',
Cminor Speicherzustand M_1' und eine Erweiterung α_1 von α, so dass

- R-values: $G, E \vdash a, M \Rightarrow v, M_1 \implies$
  $\exists\, v'.\ G', sp, L \vdash \mathcal{R}(a), E', M' \rightarrow v', E_1', M_1' \wedge \alpha_1 \vdash v \approx v'$
- L-values: $G, E \vdash a, M \stackrel{l}{\Rightarrow} loc, M_1 \implies$
  $\exists\, v'.\ G', sp, L \vdash \mathcal{L}(a), E', M' \rightarrow v', E_1', M_1' \wedge \alpha_1 \vdash ptr\ loc \approx v'$
- Anweisungen: $G, E \vdash s, M \Rightarrow out, M_1 \implies$
  $\exists\, out'.\ G', sp \vdash \mathcal{S}(s), E', M' \rightarrow out', E_1', M_1' \wedge$
  $\langle out\ und\ out'\ passende\ Terminationen \rangle$

dann finales Theorem - Korrektheit der Übersetzung:

Clight Programm p ist wohlgetypt und übersetzt zu Cminor Programm p'.
Falls $\vdash p \Rightarrow v$ und v Integer oder Fließkommazahl, dann $\vdash p' \rightarrow v$

Korrektheitsbeweis durch Simulation

Simulationsaussagen der Semantiken: ($\mathcal{R}$,$\mathcal{L}$,$\mathcal{S}$ Übersetzungsfunktionen)
Wenn $\alpha \vdash M \approx M'$, dann gibt es Cminor Umgebung E_1',
Cminor Speicherzustand M_1' und eine Erweiterung α_1 von α, so dass

- R-values: $G,E \vdash a,M \Rightarrow v,M_1 \implies$
  $\exists\, v'.\ G',sp,L \vdash \mathcal{R}(a),E',M' \rightarrow v',E_1',M_1' \wedge \alpha_1 \vdash v \approx v'$
- L-values: $G,E \vdash a,M \overset{l}{\Rightarrow} loc,M_1 \implies$
  $\exists\, v'.\ G',sp,L \vdash \mathcal{L}(a),E',M' \rightarrow v',E_1',M_1' \wedge \alpha_1 \vdash ptr\ loc \approx v'$
- Anweisungen: $G,E \vdash s,M \Rightarrow out,M_1 \implies$
  $\exists\, out'.\ G',sp \vdash \mathcal{S}(s),E',M' \rightarrow out',E_1',M_1' \wedge$
  $\langle out\ und\ out'\ passende\ Terminationen \rangle$

dann finales Theorem - Korrektheit der Übersetzung:

Clight Programm p ist wohlgetypt und übersetzt zu Cminor Programm p'.
Falls $\vdash p \Rightarrow v$ und v Integer oder Fließkommazahl, dann $\vdash p' \rightarrow v$

Backend

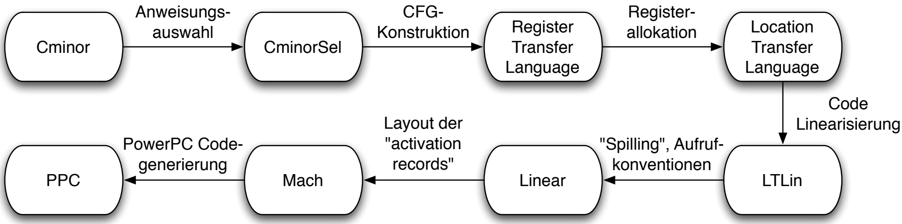
X. Leroy.
Formal certification of a compiler back-end or: programming a compiler with a proof assistant.
In *Proc. of Symposium on Principles of Programming Languages*, pp. 42–54. ACM, 2006
http://dx.doi.org/10.1145/1111037.1111042

X. Leroy.
A formally verified compiler back-end.
http://pauillac.inria.fr/~xleroy/publi/compcert-backend.pdf

Übersicht

- Quellsprache Cminor
- Zielsprache PPC (Untermenge von PowerPC, Apple bis 2006)
- “Wenn S wohldefinierte Semantik hat und C ist das Kompilat von S, dann sind S und C beobachtungsäquivalent”
- besteht aus mehreren Kompilationsschritten
- jeder Schritt kann einzeln korrekt bewiesen werden
- Korrektheit jedes Schrittes entweder
  - Verifikation der Übersetzungsfunktion, oder
  - Übersetzen, dann Resultat prüfen mit Verifizierer

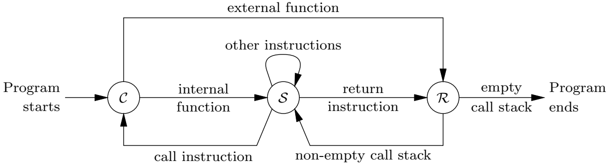
Zwischensprachen



- 6 Zwischensprachen bzw. Varianten
- jeder Übersetzungslauf hat bestimmten Zweck
- sogar kleine Optimierungen vorhanden

Semantiken der Zwischensprachen

- alle Semantiken Small Step
  (für Cminor äquivalent zu vorgestellter Big Step)
- jede Zwischensprachensemantik unterscheidet 3 Zustände:
  reguläre Zustände $\mathcal{S}$, Aufrufzustände $\mathcal{C}$ und Rückkehrzustände $\mathcal{R}$



- Erhaltung der Semantik jeweils mittels Simulation bewiesen

CminorSel: Anweisungsauswahl

CminorSel Variante von Cminor
kennt spezielle Operatoren und Adressmodi des Zielprozessors:

- $\mathtt{load}$ und $\mathtt{store}$ mit konkreten Adressmodi
- konditionale Ausdrücke c mit $\mathtt{true}$, $\mathtt{false}$, konditionalem Test $\mathtt{cond}$ und ternärem Operator $c_1\ \mathtt{?}\ a_2\ \mathtt{:}\ a_3$
- spezielle PowerPC Operatoren, z.B.
  - Integer-Operatoren mit einem unmittelbaren Operanden
    $\mathtt{addi}_n$ Addition mit n, $\mathtt{muli}_n$ Multiplikation mit n
  - Rotate-and-Mask $\mathtt{rolm}_{n,m}$
    Linksrotation um n Bits, danach logisches Und mit m

Ersetzungsregeln für Operatoren, Beispiele:

$$\begin{aligned} \mathtt{add}(e, \mathtt{intconst}(n)) &\longrightarrow \mathtt{addi}_n(e) \\ \mathtt{muli}_m(\mathtt{addi}_n(e)) &\longrightarrow \mathtt{addi}_{m\times n}(\mathtt{muli}_m(e)) \\ \mathtt{and}(e, \mathtt{intconst}(n)) &\longrightarrow \mathtt{rolm}_{0,n}(e) \end{aligned}$$

CminorSel: Anweisungsauswahl

CminorSel Variante von Cminor
kennt spezielle Operatoren und Adressmodi des Zielprozessors:

- $\mathtt{load}$ und $\mathtt{store}$ mit konkreten Adressmodi
- konditionale Ausdrücke c mit $\mathtt{true}$, $\mathtt{false}$, konditionalem Test $\mathtt{cond}$ und ternärem Operator $c_1\ \mathtt{?}\ a_2\ \mathtt{:}\ a_3$
- spezielle PowerPC Operatoren, z.B.
  - Integer-Operatoren mit einem unmittelbaren Operanden
    $\mathtt{addi}_n$ Addition mit n, $\mathtt{muli}_n$ Multiplikation mit n
  - Rotate-and-Mask $\mathtt{rolm}_{n,m}$
    Linksrotation um n Bits, danach logisches Und mit m

Ersetzungsregeln für Operatoren, Beispiele:

$$\begin{aligned}
\mathtt{add}(e,\ \mathtt{intconst}(n)) &\longrightarrow \mathtt{addi}_n(e) \\
\mathtt{muli}_m(\mathtt{addi}_n(e)) &\longrightarrow \mathtt{addi}_{m\times n}(\mathtt{muli}_m(e)) \\
\mathtt{and}(e,\ \mathtt{intconst}(n)) &\longrightarrow \mathtt{rolm}_{0,n}(e)
\end{aligned}$$

RTL und CFG

- RTL: *register transfer language* (auch *3-address code*)
- repräsentiert Funktionen als Kontrollflussgraph (CFG)
- Instruktion (Knoten) verwendet Pseudoregister (unendlicher Vorrat) und gibt Menge an potentiellen Nachfolgerknoten an
- Instruktionen: Operatoren, `goto`, `load`, `store`, `call` (Funktionsaufruf), `cond` (bedingte Verzweigung), `return`
- CFG: Map von Label nach Instruktion
- Funktionen: nicht explizit Code, sondern CFG und Einstiegspunkt

semantische Übergangsfunktion: wegen Coq funktional formuliert

aber: Übergang kann fehlschlagen! Nicht funktional formulierbar
(z.B. Referenzierung einer undeklarierten Variable)

Lösung *state-and-error monad*:
kompilierte Übergangsfunktion angewendet auf Zustand liefert entweder Fehler oder korrekten Endzustand

RTL und CFG

- RTL: *register transfer language* (auch *3-address code*)
- repräsentiert Funktionen als Kontrollflussgraph (CFG)
- Instruktion (Knoten) verwendet Pseudoregister (unendlicher Vorrat) und gibt Menge an potentiellen Nachfolgerknoten an
- Instruktionen: Operatoren, `goto`, `load`, `store`, `call` (Funktionsaufruf), `cond` (bedingte Verzweigung), `return`
- CFG: Map von Label nach Instruktion
- Funktionen: nicht explizit Code, sondern CFG und Einstiegspunkt

semantische Übergangsfunktion: wegen Coq funktional formuliert
aber: Übergang kann fehlschlagen! Nicht funktional formulierbar
(z.B. Referenzierung einer undeklarierten Variable)
Lösung *state-and-error monad*:
kompilierte Übergangsfunktion angewendet auf Zustand liefert entweder Fehler oder korrekten Endzustand

Optimierungen

echte Compiler machen viele Optimierungen in diesem Schritt
jedoch: Beweis der Semantikerhaltung von Optimierungen nichttrivial!

CompCERT zwei kleinere Optimierungen:

Konstantenpropagation: effizientere Übersetzung möglich,
wenn Operanden einer Instruktion bekannt
z.B. Verzweigungen mit konstantem Prädikat durch Goto,
Operatoren mit bekannten Operanden durch *load* ersetzen

Eliminierung von gemeinsamen Unterausdrücken: statt mehrfacher
Berechnung desselben Resultats Speicherung in Register

beide mittels Datenflussanalyse (Kildalls Worklist Algorithmus) berechnet

LTL

LTL: *location transfer language*

- weiterhin CFG und Funktionen mit Einstiegspunkt, aber Knoten *basic blocks* anstatt einzelne Instruktionen
- ersetzt Pseudoregister durch Locations: entweder
  - Hardware Prozessor Register, oder
  - *stack slots*

drei Arten von stack slots:

lokale Variablen:	`Local(`τ`,`δ`)`
eingehende Parameter:	`Incoming(`τ`,`δ`)`
ausgehende Parameter:	`Outgoing(`τ`,`δ`)`

`call` und `return` keine Register für Parameter mehr, Parameterübergabe mittels stack slots und fixen Registern entsprechend Aufrufkonventionen

LTL

LTL: *location transfer language*

- weiterhin CFG und Funktionen mit Einstiegspunkt, aber Knoten *basic blocks* anstatt einzelne Instruktionen
- ersetzt Pseudoregister durch Locations: entweder
  - Hardware Prozessor Register, oder
  - *stack slots*

drei Arten von stack slots:

lokale Variablen: `Local(`τ`,`δ`)`

eingehende Parameter: `Incoming(`τ`,`δ`)`

ausgehende Parameter: `Outgoing(`τ`,`δ`)`

`call` und `return` keine Register für Parameter mehr, Parameterübergabe mittels stack slots und fixen Registern entsprechend Aufrufkonventionen

Registerallokation

1. Lebendigkeitsanalyse: verwendet wiederum Kildalls Algorithmus
   Interferenzgraph nach Chaitin: Kanten zwischen Pseudo-Registern,
   wenn beide Werte verfügbar sein müssen
2. Färben des Interferenzgraphen:
   - benachbarte Knoten nicht gleiche Farbe
   - gleiche Farben: Werte können in gleichem Register gespeichert werden
   - liefert Mapping Φ von Pseudo-Registern auf Locations (Farben)
   - Färbprozedur selbst nicht verifiziert, aber das Ergebnis
     da Färben NP-hart, verifizieren viel einfacher
3. Coalescing (Registerverschmelzen):
   - Befehle umgeschrieben von Pseudo-Register auf Locations
   - dadurch möglich: unnötige Befehle
   - unnötige Befehle durch No-Ops ersetzen
   - Bsp: $\Phi(r) = \Phi(r')$, alter Befehl $op(\mathtt{move}, r, r', l)$
     würde zu $op(\mathtt{move}, \Phi(r), \Phi(r'), l)$, kein beobachtbares Verhalten
     deshalb Übersetzung zu $goto(l)$ (No-Op)

Registerallokation

1. Lebendigkeitsanalyse: verwendet wiederum Kildalls Algorithmus
   Interferenzgraph nach Chaitin: Kanten zwischen Pseudo-Registern,
   wenn beide Werte verfügbar sein müssen
2. Färben des Interferenzgraphen:
   - benachbarte Knoten nicht gleiche Farbe
   - gleiche Farben: Werte können in gleichem Register gespeichert werden
   - liefert Mapping Φ von Pseudo-Registern auf Locations (Farben)
   - Färbprozedur selbst nicht verifiziert, aber das Ergebnis
     da Färben NP-hart, verifizieren viel einfacher
3. Coalescing (Registerverschmelzen):
   - Befehle umgeschrieben von Pseudo-Register auf Locations
   - dadurch möglich: unnötige Befehle
   - unnötige Befehle durch No-Ops ersetzen
   - Bsp: $\Phi(r) = \Phi(r')$, alter Befehl $op(\mathtt{move}, r, r', l)$
     würde zu $op(\mathtt{move}, \Phi(r), \Phi(r'), l)$, kein beobachtbares Verhalten
     deshalb Übersetzung zu $goto(l)$ (No-Op)

Registerallokation

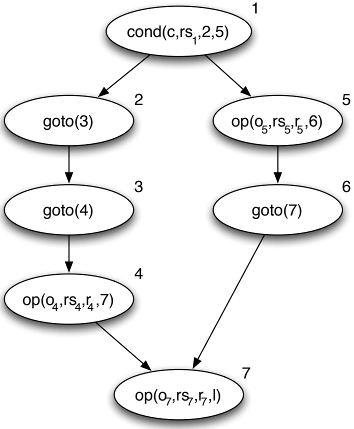
1. Lebendigkeitsanalyse: verwendet wiederum Kildalls Algorithmus
   Interferenzgraph nach Chaitin: Kanten zwischen Pseudo-Registern,
   wenn beide Werte verfügbar sein müssen
2. Färben des Interferenzgraphen:
   - benachbarte Knoten nicht gleiche Farbe
   - gleiche Farben: Werte können in gleichem Register gespeichert werden
   - liefert Mapping Φ von Pseudo-Registern auf Locations (Farben)
   - Färbprozedur selbst nicht verifiziert, aber das Ergebnis
     da Färben NP-hart, verifizieren viel einfacher
3. Coalescing (Registerverschmelzen):
   - Befehle umgeschrieben von Pseudo-Register auf Locations
   - dadurch möglich: unnötige Befehle
   - unnötige Befehle durch No-Ops ersetzen
   - Bsp: $\Phi(r) = \Phi(r')$, alter Befehl $\mathtt{op(move,}r\mathtt{,}r'\mathtt{,}l\mathtt{)}$
     würde zu $\mathtt{op(move,}\Phi(r)\mathtt{,}\Phi(r')\mathtt{,}l\mathtt{)}$, kein beobachtbares Verhalten
     deshalb Übersetzung zu $\mathtt{goto(}l\mathtt{)}$ (No-Op)

Branch tunneling und Eliminierung von No-Ops

durch Registerallokation viele No-Ops im CFG möglich

Optimierungsschritt:

- suche für jede Instruktion nächste folgende nicht-No-Op-Instruktion
- setze Nachfolgerlabel auf diese Instruktion
- eliminiere alle No-Op Knoten

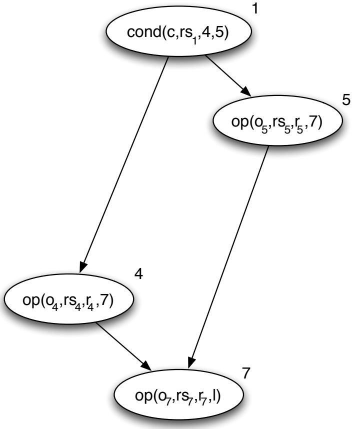


Branch tunneling und Eliminierung von No-Ops

durch Registerallokation viele No-Ops im CFG möglich

Optimierungsschritt:

- suche für jede Instruktion nächste folgende nicht-No-Op-Instruktion
- setze Nachfolgerlabel auf diese Instruktion
- eliminiere alle No-Op Knoten



Linearisierung

LTLin Variante von LTL:

- kein CFG mehr, sondern Listen von Instruktionen
- Label gibt noch zu bearbeitendes Suffix dieser Liste an

Linearisierung:

- nicht-verifizierter Caml-Code generiert Aufzählung von CFG-Labels erreichbar von Eintrittspunkt der Funktion
- Aufzählung entspricht später Instruktionsliste
- Aufzählung nachträglich validiert, so dass
  1. kein Label zweimal, aber
  2. alle vom Eintrittspunkt erreichbaren Labels

  in Aufzählung vorhanden (mittels Datenflussanalyse geprüft)

Linearisierung

LTLin Variante von LTL:

- kein CFG mehr, sondern Listen von Instruktionen
- Label gibt noch zu bearbeitendes Suffix dieser Liste an

Linearisierung:

- nicht-verifizierter Caml-Code generiert Aufzählung von CFG-Labels erreichbar von Eintrittspunkt der Funktion
- Aufzählung entspricht später Instruktionsliste
- Aufzählung nachträglich validiert, so dass
  1. kein Label zweimal, aber
  2. alle vom Eintrittspunkt erreichbaren Labels

  in Aufzählung vorhanden (mittels Datenflussanalyse geprüft)

Linear: `spill`, `reload` und Aufrufkonventionen

Linear Variante von LTLin: statt beliebigen Locations Maschinenregister

1. *spill* und *reload*:
   - Locations aus Registerallokationsphase entweder stack slots oder Register
   - jetzt nur noch Register verfügbar, also Anpassungen nötig

     spill: explizites Auslagern eines Registerwert in den Speicher
     reload: Wert aus Speicher in Register holen
2. Herstellen der Aufrufkonventionen:
   durch zusätzliche move-Befehle Operanden an die für Parameterübergabe benötigten Stellen schieben

Linear: *spill*, *reload* und Aufrufkonventionen

Linear Variante von LTLin: statt beliebigen Locations Maschinenregister

1. *spill* und *reload*:
   - Locations aus Registerallokationsphase entweder
     stack slots oder Register
   - jetzt nur noch Register verfügbar, also Anpassungen nötig

     spill: explizites Auslagern eines Registerwert in den Speicher
     reload: Wert aus Speicher in Register holen

2. Herstellen der Aufrufkonventionen:
   durch zusätzliche move-Befehle Operanden an die für Parameterübergabe benötigten Stellen schieben

Linear: *spill*, *reload* und Aufrufkonventionen

Linear Variante von LTLin: statt beliebigen Locations Maschinenregister

1. *spill* und *reload*:
   - Locations aus Registerallokationsphase entweder stack slots oder Register
   - jetzt nur noch Register verfügbar, also Anpassungen nötig

     spill: explizites Auslagern eines Registerwert in den Speicher

     reload: Wert aus Speicher in Register holen
2. Herstellen der Aufrufkonventionen:
   durch zusätzliche move-Befehle Operanden an die für Parameterübergabe benötigten Stellen schieben

Idee der Übersetzung

- $[\![.]\!]$ ist Übersetzungsfunktion, l, ls Locations, r, rs Register
- $\mathtt{reg\_for}(l)$: $\mathtt{if}\ l\ \mathtt{Register\ then}\ l\ \mathtt{else\ temporäres\ Register}\ r'$
- $\mathtt{regs\_for}(ls)$ und $\mathtt{reloads}(ls, rs)$: Liftings auf Listen
- $\mathtt{parallel\_move}$: mehrere gleichzeitige move-Befehle
- $\mathtt{loc\_arguments}(sig)$ und $\mathtt{loc\_result}(sig)$:
  bestimmen Locations für Parameterübergabe der Aufrufkonvention
- $\mathtt{spill}$ und $\mathtt{reload}$:
  $$[\![op(op, ls, l)\#c]\!] \equiv \mathtt{let}\ rs = \mathtt{regs\_for}(ls);\ r = \mathtt{reg\_for}(l)\ \mathtt{in}$$
  $$\mathtt{reloads}(ls, rs)\#op(op, rs, r)\#\mathtt{spill}(r, l)\#[\![c]\!]$$
- Aufrufkonventionen:
  $$[\![\mathtt{call}(sig, id, ls, l)\#c]\!] \equiv \mathtt{parallel\_move}(ls, \mathtt{loc\_arguments}(sig))\#$$
  $$\mathtt{call}(sig, id)\#\mathtt{spill}(\mathtt{loc\_result}(sig), l)\#[\![c]\!]$$

Idee der Übersetzung

- $[\![.]\!]$ ist Übersetzungsfunktion, l, ls Locations, r, rs Register
- $\mathtt{reg\_for}(l)$: $\mathtt{if}\ l\ \mathtt{Register\ then}\ l\ \mathtt{else\ temporäres\ Register}\ r'$
- $\mathtt{regs\_for}(ls)$ und $\mathtt{reloads}(ls, rs)$: Liftings auf Listen
- $\mathtt{parallel\_move}$: mehrere gleichzeitige move-Befehle
- $\mathtt{loc\_arguments}(sig)$ und $\mathtt{loc\_result}(sig)$:
  bestimmen Locations für Parameterübergabe der Aufrufkonvention
- $\mathtt{spill}$ und $\mathtt{reload}$:

$$\begin{aligned}[\![op(op, ls, l)\#c]\!] \equiv\ & \mathtt{let}\ rs = \mathtt{regs\_for}(ls);\ r = \mathtt{reg\_for}(l)\ \mathtt{in} \\ & \mathtt{reloads}(ls, rs)\#op(op, rs, r)\#\mathtt{spill}(r, l)\#[\![c]\!]\end{aligned}$$

- Aufrufkonventionen:

$$\begin{aligned}[\![\mathtt{call}(sig, id, ls, l)\#c]\!] \equiv\ & \mathtt{parallel\_move}(ls, \mathtt{loc\_arguments}(sig))\# \\ & \mathtt{call}(sig, id)\#\mathtt{spill}(\mathtt{loc\_result}(sig), l)\#[\![c]\!]\end{aligned}$$

Idee der Übersetzung

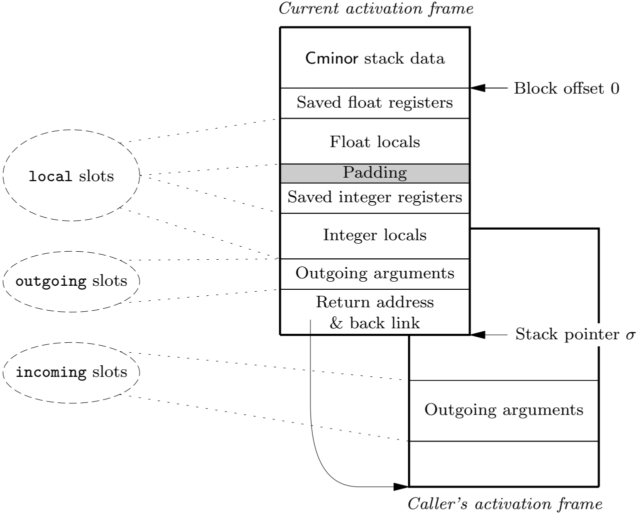
- $[\![.]\!]$ ist Übersetzungsfunktion, l, ls Locations, r, rs Register
- $\mathtt{reg\_for}(l)$: $\mathtt{if}\ l\ \mathtt{Register\ then}\ l\ \mathtt{else\ tempor\ddot{a}res\ Register}\ r'$
- $\mathtt{regs\_for}(ls)$ und $\mathtt{reloads}(ls, rs)$: Liftings auf Listen
- $\mathtt{parallel\_move}$: mehrere gleichzeitige move-Befehle
- $\mathtt{loc\_arguments}(sig)$ und $\mathtt{loc\_result}(sig)$:
  bestimmen Locations für Parameterübergabe der Aufrufkonvention
- $\mathtt{spill}$ und $\mathtt{reload}$:
  $$\begin{aligned}[\![op(op, ls, l)\#c]\!] \equiv\ & \mathtt{let}\ rs = \mathtt{regs\_for}(ls);\ r = \mathtt{reg\_for}(l)\ \mathtt{in} \\ & \mathtt{reloads}(ls, rs)\#op(op, rs, r)\#\mathtt{spill}(r, l)\#[\![c]\!]\end{aligned}$$
- Aufrufkonventionen:
  $$\begin{aligned}[\![\mathtt{call}(sig, id, ls, l)\#c]\!] \equiv\ & \mathtt{parallel\_move}(ls, \mathtt{loc\_arguments}(sig))\# \\ & \mathtt{call}(sig, id)\#\mathtt{spill}(\mathtt{loc\_result}(sig), l)\#[\![c]\!]\end{aligned}$$

Mach

- bisher 3 unendliche Vorräte an stack slots: *Local*, *Incoming*, *Outgoing*
  jetzt: Mapping der stack slots auf konkrete Speicherstellen
  - *Local* und *Incoming* werden Speicherstellen im stack frame der aufrufenden Methode
  - *Outgoing* werden Speicherstellen im stack frame der aufgerufenen Methode
- jede Funktion zwei neue Byte-Offsets:

  retaddr: Offset der Rückkehradresse im activation record
  link: Rückwärtsverweis auf den activation record der aufrufenden Methode
- keine automatische Wiederherstellung von Registern bei Rückkehr aus einer Funktion
  durch explizite Befehle muss sichergestellt werden, dass Werte bei Rückkehr aus einer Funktion wieder an der vorherigen Stelle liegen

Layout der activation records



Am Ziel: PPC

- abstrakte Syntax für Untermenge von PowerPC Assemblercode
- beinhaltet 82 der über 200 Instruktionen
- zusätzlich 7 Makroinstruktionen: kombinieren Instruktionen
  diese Abstraktion nötig, da Speicherabstraktion sonst zu schwach, Folge der Instruktionen nicht verifizierbar
- wenn externe Funktionen deterministisch sind, dann PPC auch
- Übersetzungsschritt umfangreich, aber nicht kompliziert

Resultate

- $\sim$ 35000 Zeilen Coq + $\sim$ 1500 Caml code
- Evaluierung:
  - Übersetzungszeit: zwischen 50% und 200% der Zeit von `gcc -O1` (gcc mit Optimierungen 1.Stufe)
  - Performanz übersetzter Programme: nur kleine Testsuite deuten an: deutlich besser als `gcc -O0`, vergleichbar mit `gcc -O1` (allerdings Ausreißer)
- Probleme:
  - funktioniert nicht mit fest vorgegebenen Speicherlimits, Übersetzungsschritte können Speicherbedarf erhöhen
  - SSA (*static single assignment*) wäre besser für Optimierungen jedoch Semantik nicht-trivial, da SSA-Eigenschaft nicht-lokal