Semantik hardware-naher Programme

Hendrik Tews

Institut für Theoretische Informatik, TU Dresden
`tews@tcs.inf.tu-dresden.de`

Inhalt

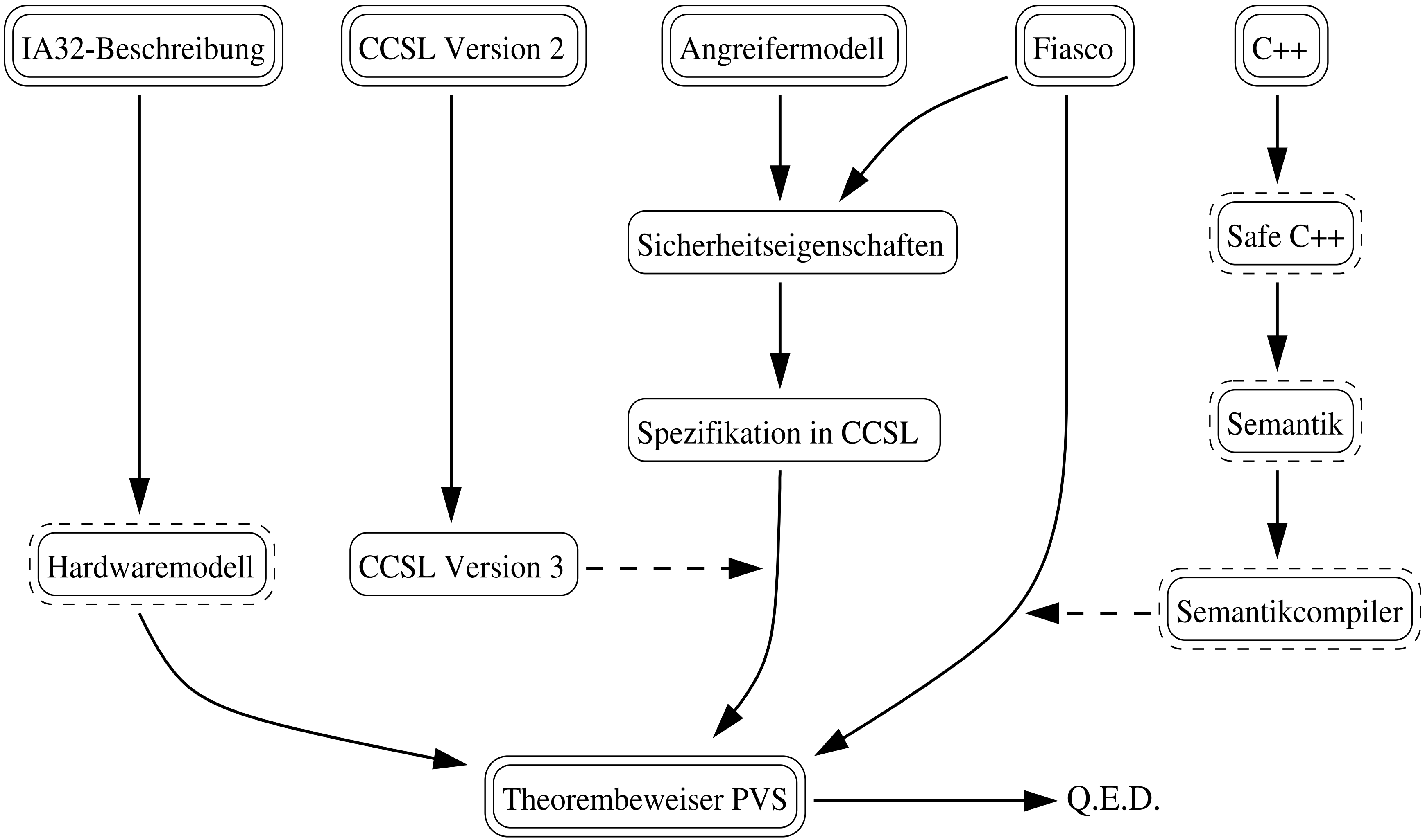
Fiasco

- L4 kompatibler Mikrokern der zweiten Generation
- im DROPS Projekt entwickelt (Dresden Real-Time Operating System)
- 15.000 Zeilen C++ (ohne internen Debugger)
- Echtzeit: Video unter Volllast
- Sicherheit: Signieren trotz Vieren

VFiasco

- Verifikation einiger Eigenschaften von Fiasco auf Quelltextebene
  - interner Page-fault-handler stürzt nicht ab
  - interne Speicherverwaltung ist korrekt
- Semantik für hardware-nahe Programmidiome
  - Kontrollfluss `break`, `goto`, `longjmp`
  - Typkonvertierungen (`unsigned` $\Longleftrightarrow$ `void *`)
  - Hardwaremanipulationen

VFiasco Projekt Überblick



Als Beispiel

- Gegenseitiger Ausschluss [1, 2]

Model checking

- Fluke's IPC [3]

Statische Checks ala lint [4]

Theorembeweisen

- Ensemble (Netzwerk Protokollstacks) [5]
- Java Bytecode Verifizierer [6, 7]

[1] E. A. Emerson. Temporal and modal logic. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 995–1072. Elsevier Science Publishers, Amsterdam, The Netherlands, 1990.

[2] B. Jacobs. Exercises in coalgebraic specification. In R. Crole R. Backhouse and J. Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, volume 2297 of *Lecture Notes in Computer Science*, pages 237–280. Springer, Berlin, 2002.

[3] P. Tullmann, J. Turner, J. McCorquodale, J. Lepreau, A. Chitturi, and G. Back. Formal methods: A practical tool for OS implementors. In *Workshop on Hot Topics in Operating Systems*, pages 20–25, 1997.

[4] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Symposium on Operating Systems Design and Implementation (OSDI 2000)*, San Diego, CA, 23–25 October 2000.

[5] X. Liu, C. Kreitz, R. van Renesse, J. Hickey, M. Hayden, K. P. Birman, and R. L. Constable. Building reliable, high-performance communication systems from components. In *17th ACM Symposium on Operating System Principles (SOSP)*, pages 80–92, Kiawah Island, SC, December 1999.

[6] C. Pusch. Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL. In W. Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, volume 1579 of *Lecture Notes in Computer Science*, pages p. 89–103. Springer-Verlag, 1999.

[7] G. Klein and T. Nipkow. Verified bytecode verifiers. *Theoretical Computer Science*, 298:583–626, 1003.

Typumwandlungen (Casts)

- Bitmuster wird in einem anderen Typ interpretiert
  - für Hardware-Datentypen
  - mit `char` zum Kopieren
- `unsigned int` $\Longleftrightarrow$ `void *`
- Umwandlungen zwischen verschiedenen Pointertypen

Strukturen / Unions

- eventuell nur teilweise initialisiert
- Padding
- ... ist machmal wesentlich

```
union { int i; char c; } u;
u.i = -1;
u.c = 1;
```

Zugriff auf nichtexistenten Speicher

```
T a = * reinterpret_cast<T *>(50)
```

Datentypen im Standard nicht voll spezifiziert

- Bitbreite der Integertypen (≥ 7), Einer/Zweierkomplement
- Reihenfolge der Felder in Objekten
- Pointer sind 4/8/12 Bytes lang

Typ void

Schlussfolgerungen

- ungetypter Speicher (im Gegensatz zu $\mathit{heap} : \mathit{address} \longrightarrow \mathit{value}$)
- Allgemeine aber skalierbare Semantik
  z.B. für 8 Bit `char` und 32 Bit `unsigned`

C++ Standard

- beschreibt Objekte (d.h. Speicherrepräsentationen von Werten)
- Objekte haben eine Adresse, einen Typ und okkupieren Speicher
- unterscheidet zwischen Werten, Wertepräsentationen und Objektrepräsentationen
- Lesezugriffe funktionieren nur, auf *gültigen* Objektrepräsentationen

Schlussfolgerungen

- der Standard gestattet Typüberprüfung zur Laufzeit (wie in Lisp)
- um Typsicherheit zu beweisen, reicht es aus, normale Termination für jede mögliche C++ Implementation zu beweisen

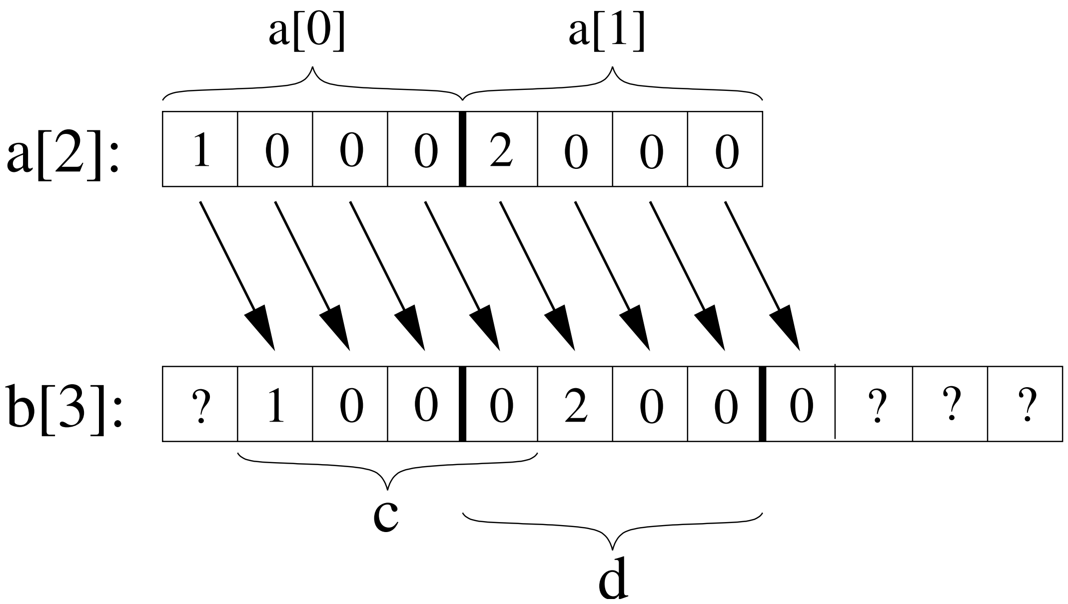
Ansatz für eine Semantik der Datentypen

- Wertebereiche nur für einfache Typen (nicht für komposite)
- Semantik folgt dem Standard
- stark unterspezifiziert
- hat jedoch viele Modelle (z.B. für i386 gcc)

```
unsigned a[2] = { 1, 2 };
unsigned b[3];

memcpy((char *)(b) + 1, a, sizeof(a));
unsigned c = * (unsigned *) ((char *)(b) + 1);     //  √

unsigned d = b[1];                                 //  ↯
```



Allgemeine Struktur für Basistypen

```
Data_type_structure : Type = [#
  size : nat,
  to_byte : [Data, Address -> list[Byte]],
  from_byte : [list[Byte], Address -> lift[Data]]
#]

data_type? : PRED[Data_type_structure] = Lambda(ct : Data_type_structure) :
  (Forall(d : Data, a : Address) : length(to_byte(ct)(d,a)) = size(ct))
     And
  (Forall(d : Data, a : Address) : up?(from_byte(ct)(to_byte(ct)(d,a), a)))
     And
  (Forall(d : Data, a : Address) : down(from_byte(ct)(to_byte(ct)(d,a), a)) = d )
     And
  (Forall(l : list[Byte], a : Address) : up?(from_byte(ct)(l, a)) Implies length(l) = size(ct))
     And
  (Forall(d : Data, a1,a2 : Address) : up?(from_byte(ct)(to_byte(ct)(d,a1), a2)))
```

Semantik

```
Semantics_bool: Type = bool
dt_bool_exists : Axiom Exists (x: (data_type?[Semantics_bool])): true
dt_bool : (data_type?[Semantics_bool])
```

Modell

```
dt_bool : (data_type?[Semantics_bool]) = (#
  size := 1,
  to_byte := Lambda(b : bool, a : Address) : IF b Then (: 1 :) Else (: 0 :) Endif,
  from_byte := Lambda(bl : list[Byte], a : Address) :
                 IF bl = (: 1 :) Then up(true)
                 Elsif bl = (: 0 :) Then up(false)
                 Else bottom   Endif   #)
```

Datentypen Zusammenfassung

- keine Gleitkommazahlen
- Semantik für Datentypen mit Casts und Pointerarithmetik
- Wertmenge nur für einfache Typen (`char`, `unsigned`, `...`)
- keine Wertmenge für komplexe Typen
- Semantik beschreibt die Speicherrepräsentation und die Speicherzugriffe
- Normale Termination $\Longrightarrow$ Typsicherheit

```
while( ... ){
  if( ... ) continue;
  if( ... ) break;
  if( ... ) return;
}
```

Ähnliche Phänomene

- Ausnahmen (exceptions)
- `longjmp/setjmp`
- nicht terminierende Programmteile

Lösung von Jacobs/Huismann im LOOP Projekt für Java [1, 2]

- Unterscheidung zwischen *normaler* und *abrupter* Termination
- Programmteile sind Funktionen $State \longrightarrow Result[State]$

Im LOOP Projekt nicht behandelt

- `goto`
- `switch` von C (berechnete Sprünge in einen anderen Block hinein)

[1] M. Huisman and B. Jacobs. Java program verification via a Hoare logic with abrupt termination. In T. Maibaum, editor, *FASE 2000 Proc.*, volume 1783 of *LNCS*, pages 284–303. Springer, 2000.

[2] M. Huisman and B. Jacobs. Inheritance in higher order logic: Modeling and reasoning. In M. Aagaard and J. Harrison, editors, *TPHOLs 2000*, volume 1869 of *LNCS*, pages 301–319.

- Programmteile sind Funktionen $\quad State \longrightarrow Result[State]$
- *State* ist die Menge aller Speicher- oder Maschinenzustände
- $Result[State]$ ist eine disjunkte Vereinigung

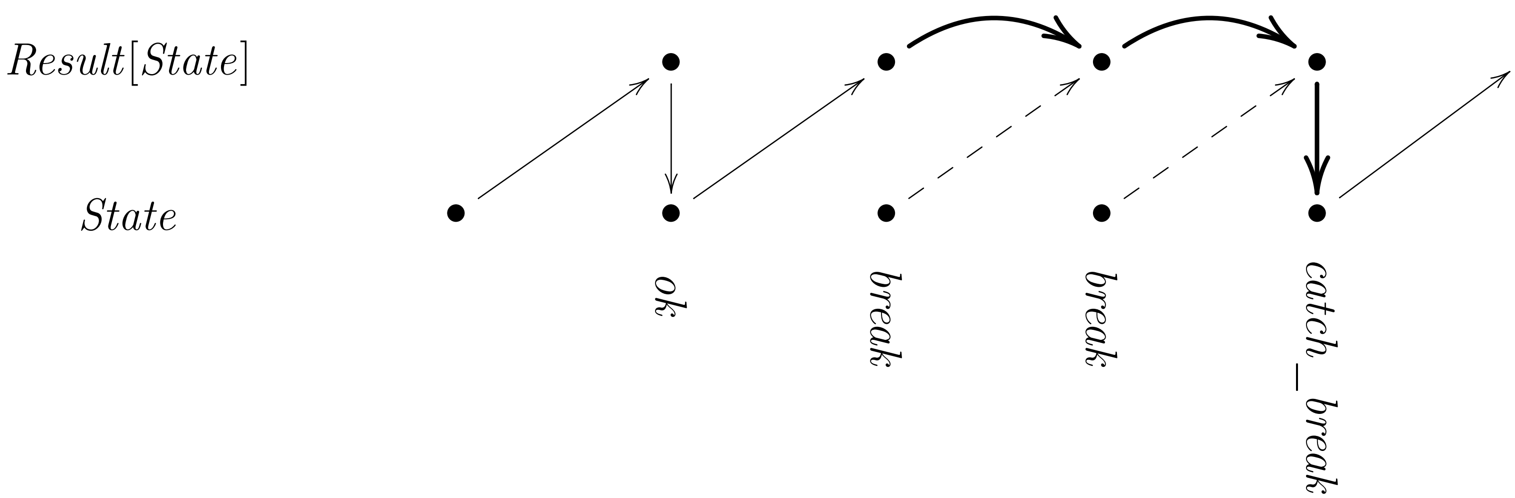
$$Result[State] \quad = \quad \left\{ \begin{array}{ll} State \uplus & ok \;\; \text{(normale Termination)} \\ State \times String \uplus & continue \\ State \times String \uplus & break \\ State \uplus & return \\ \{*\} & fail \;\; \text{(Nichttermination)} \end{array} \right.$$

- Komposition von Anweisungen / Programmteilen:

$$(p\ ;\ q)(s_1) \quad = \quad \left\{ \begin{array}{ll} q\, s_2 & \text{falls } p\, s_1 = ok(s_2) \\ p\, s_1 & \text{sonst} \end{array} \right.$$

- Wiederaufsetzen mit speziellen Funktionalen

$$catch\_break\ p\ s_1 \quad = \quad \left\{ \begin{array}{ll} ok(s_2) & \text{falls } p\, s_1 = ok(s_2) \\ ok(s_2) & \text{falls } p\, s_1 = break(s_2) \\ p\, s_1 & \text{sonst} \end{array} \right.$$



Result[State]
State
ok
break
break
catch_break

- Jacobs & Huismann verwenden unterschiedliche Typen für die Abnormitäten von Befehlen und Ausdrücken
- hier einfach eine Suppe
- Programmteile sind Funktionen $\quad State \longrightarrow Result[State, Value]$

  für Befehle $Value = \{*\}$
- in Pvs

```
Result[State, Value : Type] : Datatype
Begin
  ok(next : State, value : Value) : ok?
  break(next : State) : break?
  continue(next : State) : continue?
  return(next : State) : return?
  hang : hang?
  fail : fail?
End Result
```

Strukturierte Sprünge

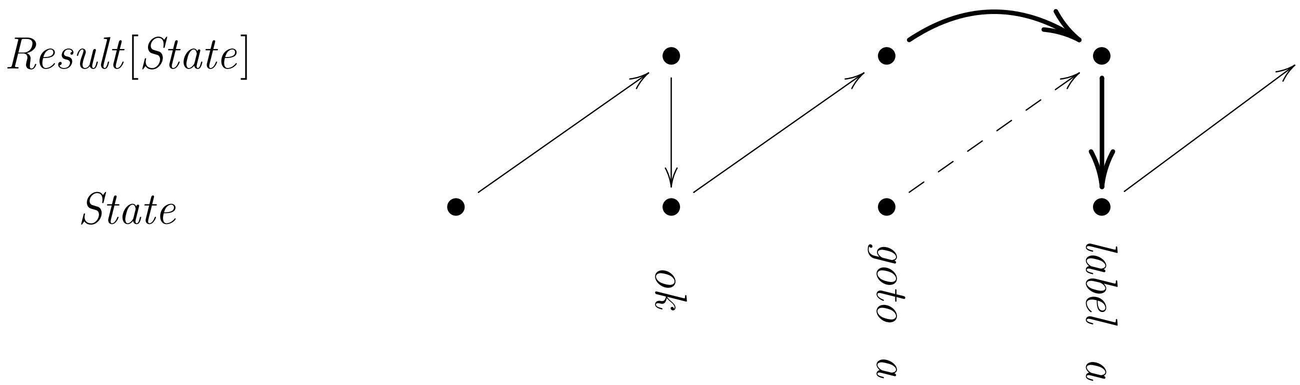
- Sprungmarken haben begrenzte Sichtbarkeit (umschließende Funktion)
- Sprünge in eine andere Funktion gibt es nicht
- + weitere Einschränkungen

`switch`

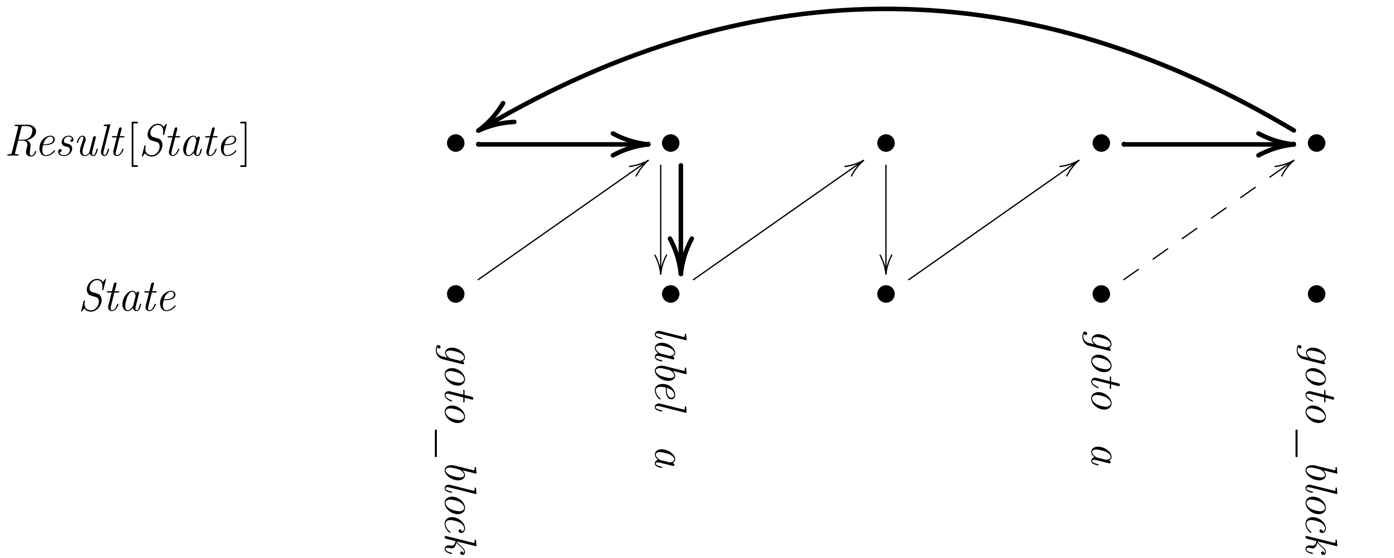
- berechnete Sprünge (im Gegensatz zu Fallunterscheidung)
- durchfallen (ohne explizites `break`)
- Sprungmarken (`case`) können in tiefer geschachtelten Blöcken liegen (Duff's device)

```
int rounds = count / 8;
switch(count % 8){
case 0: while(rounds-- > 0){ *to++ = *from++;
case 7:                      *to++ = *from++;
case 6:                      *to++ = *from++;
case 5:                      *to++ = *from++;
case 4:                      *to++ = *from++;
case 3:                      *to++ = *from++;
case 2:                      *to++ = *from++;
case 1:                      *to++ = *from++;
                 }           };
```

Vorwärtsspringen



Rückwärtssprünge



Semantik für „schmutzige“ Programme

- Typumwandlungen (casts)
- abrupte Termination
- goto, switch

Andere Ansätze

- Objektcodeverifikation [1]
- ASM Semantik [2, 3]
- nichtdeterministische C Semantik in HOL [4, 5]

[1] M. Wahab. Verification and abstraction of flow-graph programs with pointers and computed jumps. Research Report CS-RR-354, Dep. of Comp. Sci., University of Warwick, 1998.

[2] Y. Gurevich and J. Huggins. The Semantics of the C Programming Language. In E. Börger, H. Kleine Büning, G. Jäger, S. Martini, and M. M. Richter, editors, *Computer Science Logic*, volume 702 of *LNCS*, pages 274–309. Springer, 1993.

[3] C. Wallace. The Semantics of the C++ Programming Language. In E. Börger, editor, *Specification and Validation Methods*, pages 131–164. Oxford University Press, 1995.

[4] M. Norrish. Deterministic expressions in c. In D. S. Swierstra, editor, *Programming Languages and Systems; ESOP '99 Proceedings*, volume 1675 of *LNCS*, pages 147–161. Springer, 1999.

[5] M. Norrish. *Formalising C in HOL*. PhD thesis, University of Cambridge, 1998.