Semantics of system-level programs

Hendrik Tews

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

presented by Harvey Tuch

Contents

I. General VFiasco

VERIFIED

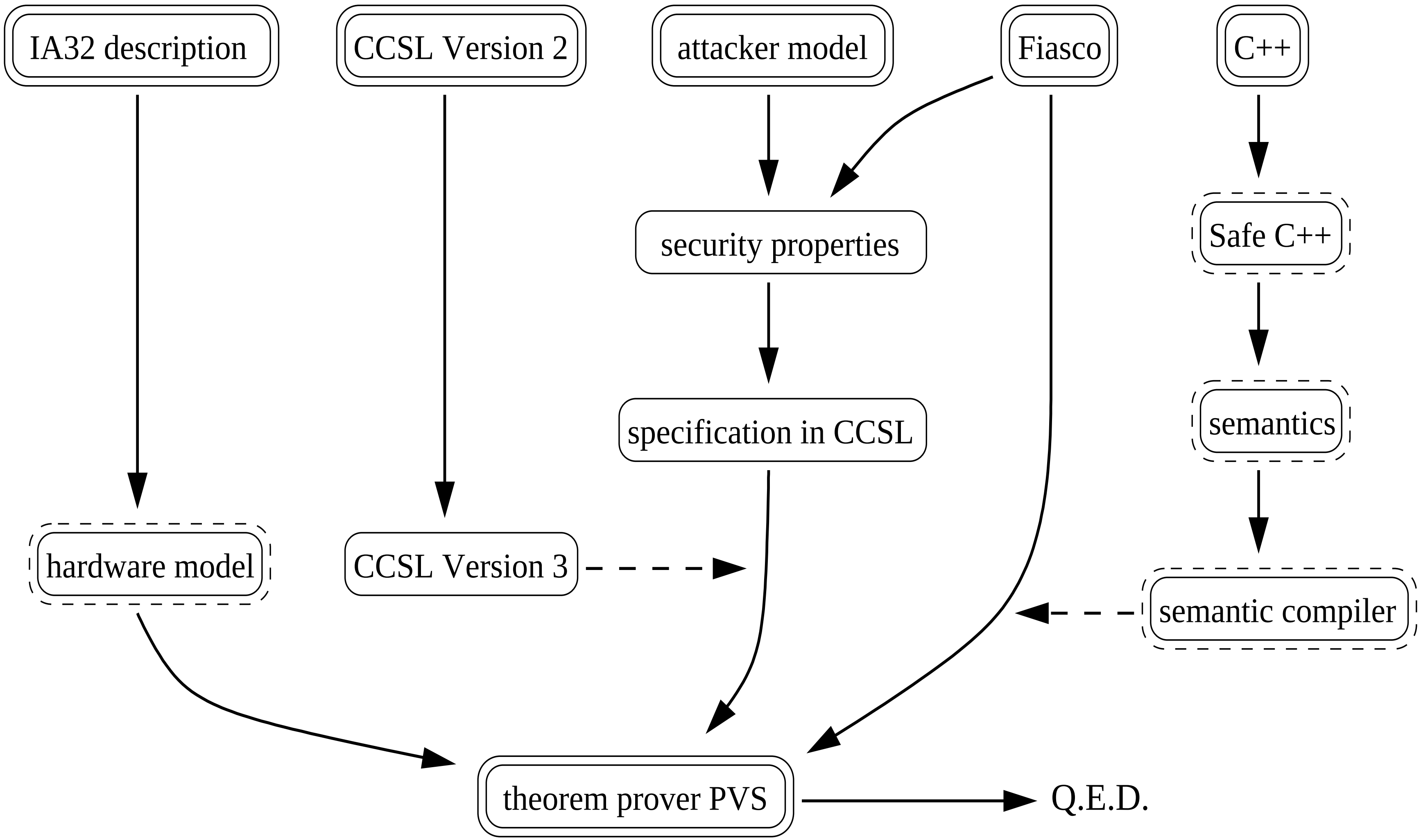
FIASCO

Fiasco

- 2nd generation micro kernel, compatible with L4
- developed in the DROPS project
  (Dresden Real-Time Operating System)
- 15.000 lines of C++ (without kernel debugger)
- real time: view movies with load 10
- security: sign with newest viruses present

VFiasco

- Verification of some properties of Fiasco
  (on source code level)
  - internal page-fault handler terminates
  - internal memory management is correct
- Semantics for system-level programming
  - control flow `break`, `goto`, `longjmp`
  - type casts (`unsigned` $\Longleftrightarrow$ `void *`)
  - hardware manipulations


IA32 description
CCSL Version 2
attacker model
Fiasco
C++
security properties
Safe C++
specification in CCSL
semantics
hardware model
CCSL Version 3
semantic compiler
theorem prover PVS
Q.E.D.

As example in theoretical work

- mutal exclusion [1, 2]

Model checking

- Fluke's IPC [3]

static checks (lint like) [4]

theorem proving

- Ensemble (network protokoll stacks) [5]
- Java bytecode verifier [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.

II. Semantics of C++ data types

type casts

- interpreting some bits in a different type
  - needed for hardware data types
  - for copying (via `char`)
- `unsigned int` $\Longleftrightarrow$ `void *`
- casts between different pointer types

structures / unions

- maybe only partially initialized
- padding
- ... is sometimes essential

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

Access non-existent memory

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

data types are not fully specified in the standard

- width of intergertypes (≥ 7), one's or two's complement
- layout of fields in objects
- pointers are 4/8/12 bytes long

Type void

Conclusion

- untyped memory (in contrast to $heap : address \longrightarrow value$)
- general, scalable semantics
  e.g. for 8 bit `char` and 32 bit `unsigned`

C++ Standard

- describes objects (i.e. memory representations of values)
- objects have an address, a type and occupy memory
- standard distinguishes between values, the value representation and the object representation
- reading in memory works only at addresses with valid object representations

Conclusion

- standard permits runtime type checking (a la lisp)
- to prove type safety, it is sufficient to prove termination without errors for every possible C++ implementation

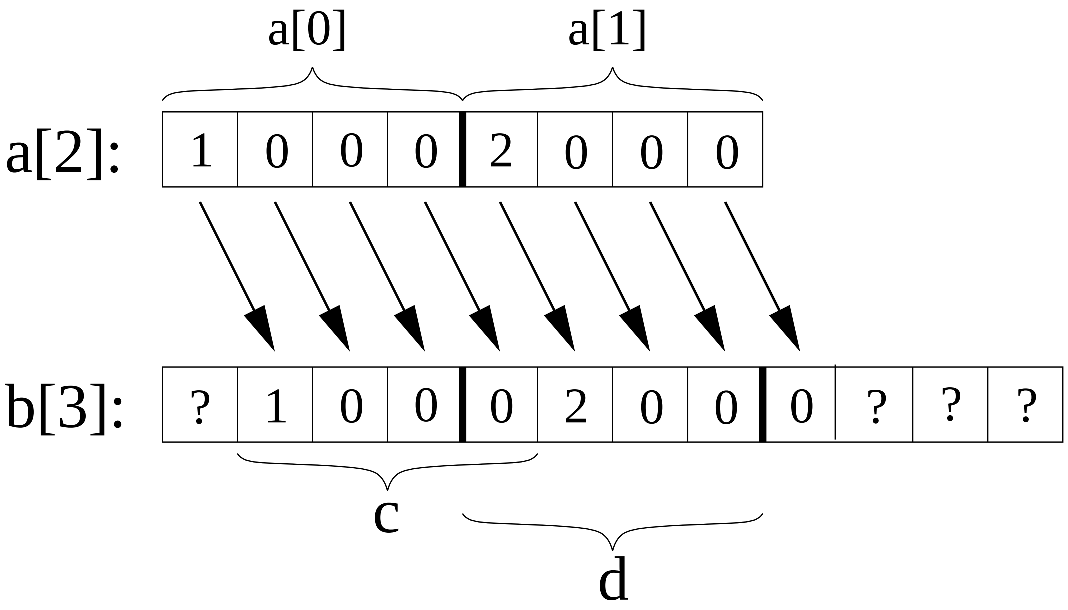
Approach for a semantics of data types

- define sets of values only for simple types (not for composite ones)
- semantics follows the standard
- semantics is quite loose (leaving many details open)
- has many models (e.g. for 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];                                  //  ↯
```



General structure for base types

```
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)))
```

Semantics

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

Model

```
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    #)
```

- no floating point numbers
- semantics for data types with casts and pointer arithmetics
- domain of values only for simple types (`char, unsigned, ...`)
- no domain of values for complex types
- semantics describes memory representation and memory access
- normal termination $\Longrightarrow$ type safety

[thanks to Harvey Tuch, member of the VFiasco data type task force in autumn '03]

III. Semantics for abrupt termination

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

Similar phenomena

- exceptions
- `longjmp/setjmp`
- non-termination

Solution of Jacobs/Huismann in the LOOP project for Java [1, 2]

- distinguish *normal* and *abrupt* termination
- (parts of) programs are modelled as functions $State \longrightarrow Result[State]$

Not treated in the LOOP project

- `goto`
- C's `switch` (computed jumps, possibly inside other blocks)

[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.

- (parts of) programs are functions $\quad State \longrightarrow Result[State]$
- $State$ is the set of all memory or computer states
- $Result[State]$ is a disjoint union

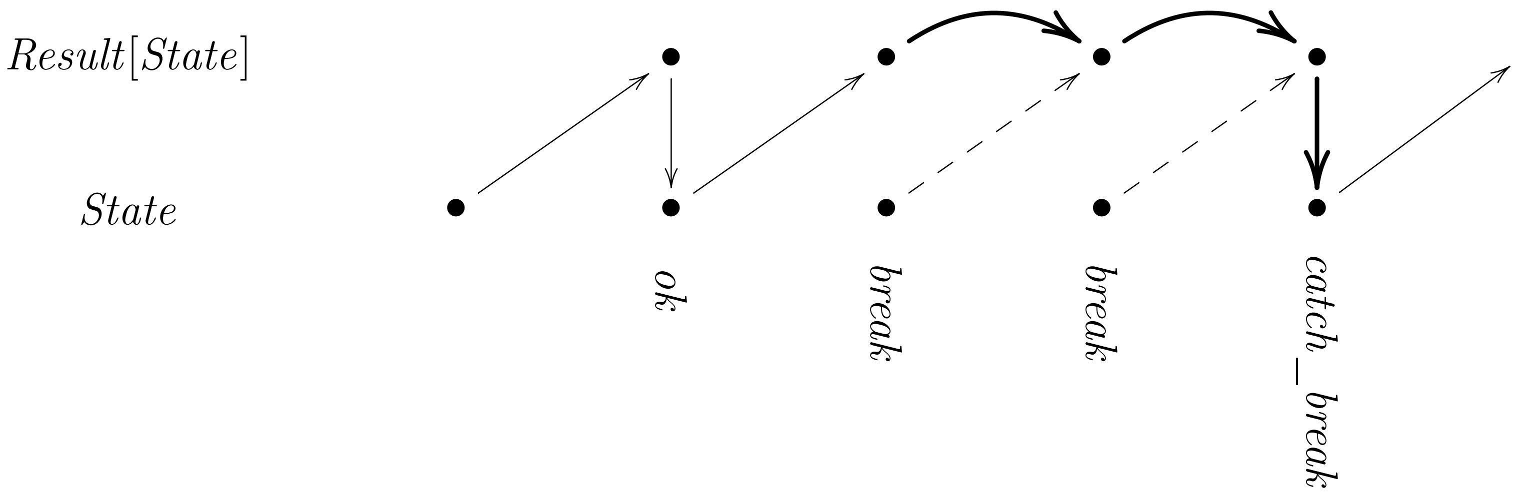
$$Result[State] \quad = \quad \begin{cases} State \uplus & ok \quad \text{(normal termination)} \\ State \times String \uplus & continue \\ State \times String \uplus & break \\ State \uplus & return \\ \{*\} & fail \quad \text{(non-termination)} \end{cases}$$

- sequentiel composition of statements

$$(p\ ;\ q)(s_1) \quad = \quad \begin{cases} q\,s_2 & \text{if } p\,s_1 = ok(s_2) \\ p\,s_1 & \text{otherwise} \end{cases}$$

- continue execution with special functions

$$catch\_break\ p\ s_1 \quad = \quad \begin{cases} ok(s_2) & \text{if } p\,s_1 = ok(s_2) \\ ok(s_2) & \text{if } p\,s_1 = break(s_2) \\ p\,s_1 & \text{otherwise} \end{cases}$$



$Result[State]$
$State$
ok
$break$
$break$
$catch\_break$

- Jacobs & Huismann use different types for the abnormalities of statements and expressions
- here only one type
- (parts of) programs are functions $State \longrightarrow Result[State, Value]$

  for statements $Value = \{*\}$
- 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
```

IV. Switch & Goto

structured jumps

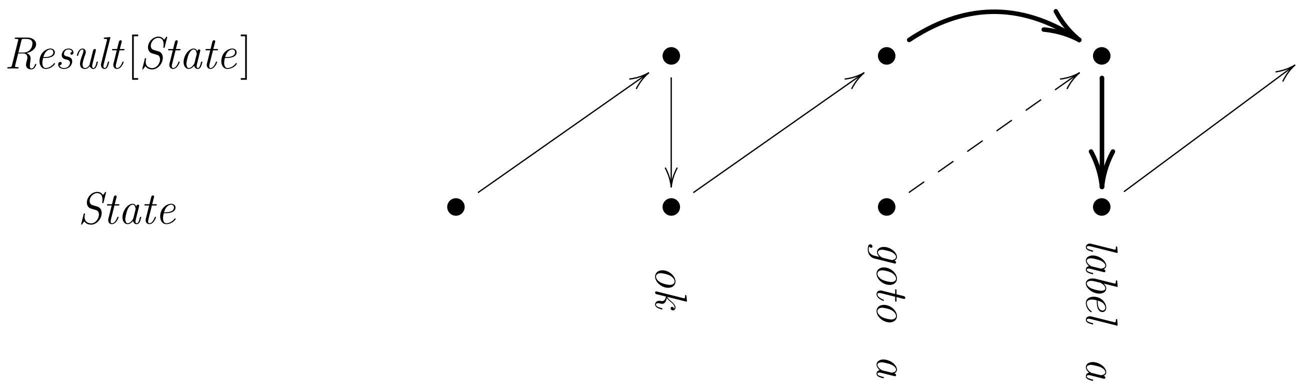
- jump labels have a limited scope (function body)
- no jumps into other functions
- + other restrictions

switch

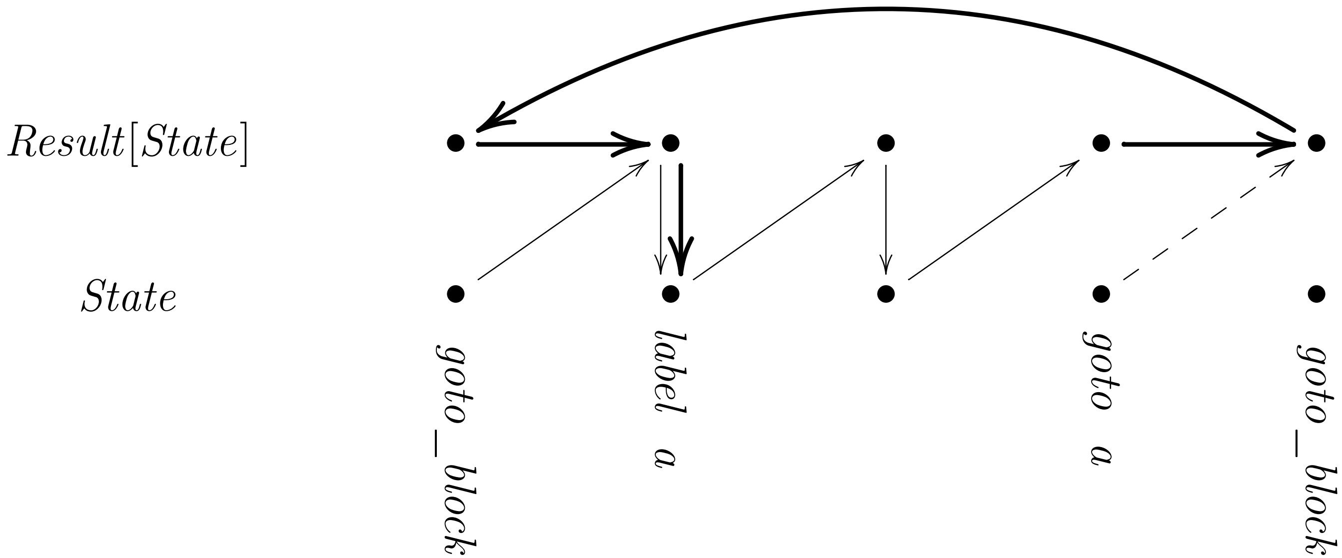
- computed jump (in contrast to a case switch)
- fall through (without `break`)
- labels may appear inside other blocks (Duff's device)

```
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++;
                  }            };
```

forward jumps



backward jumps



Translate source code into PVS

```
rounds : posnat = 1
i : posnat = 2

duff(source, dest : posnat, count : nat) : [Result[State, Unit] -> Result[State,Unit]] =
  write_int(rounds, div(const_int(count), const_int(8))) ##
  write_int(i, const_int(0)) ##
  int_switch_stm( rem( const_int(count), const_int(8)),
      int_case(0) ##
              gwhile_stm(
                 const_int(0) < post_decr_const(rounds),
                     write_int_array(dest, read_int(i), read_int_array(source, read_int(i))) ##
                     write_int(i, read_int(i) ++ const_int(1)) ##
      int_case(7) ## write_int_array(dest, read_int(i), read_int_array(source, read_int(i))) ##
                     write_int(i, read_int(i) ++ const_int(1)) ##
      int_case(6) ## write_int_array(dest, read_int(i), read_int_array(source, read_int(i))) ##
                     write_int(i, read_int(i) ++ const_int(1)) ##
          .................
      int_case(2) ## write_int_array(dest, read_int(i), read_int_array(source, read_int(i))) ##
                     write_int(i, read_int(i) ++ const_int(1)) ##
      int_case(1) ## write_int_array(dest, read_int(i), read_int_array(source, read_int(i))) ##
                     write_int(i, read_int(i) ++ const_int(1))
              )  % end gwhile_stm
  ) % end int_switch_stm
```

Proof it

```
duff_total : Lemma
  Forall(s : State, source, dest : posnat, count : nat) :
    duff_var_ok(source, dest, count,s) Implies
      duff(source, dest, count)(ok(s,unit)) =
      ok(s WITH [
          `vars := Lambda(j : posnat) :
              IF j = rounds Then int(−1)
              Elsif j = i Then int(count)
              Elsif cell_in_array(s, dest)(j) And
                      index_from_cell(s, dest, j) < count
                  Then
                      s`vars(get_array(s`vars(source))
                              `fields(index_from_cell(s, dest, j)))
              Else s`vars(j)
              Endif
          ], unit)
```

Working Semantics for "dirty" programs

- type casts
- abrupt termination
- goto, switch
- simple compositional denotational semantics (no continuations)
- Duff's device verified

other approaches

- object code verification [1]
- ASM Semantics [2, 3]
- non-deterministic semantics of C 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.