

„Hilberts Albtraum“

(The answer may *not* be out there!)

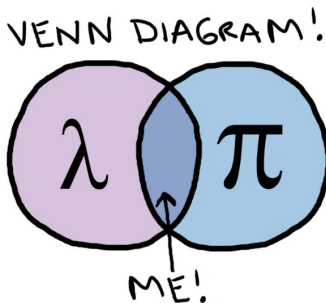
Jonas Betzendahl

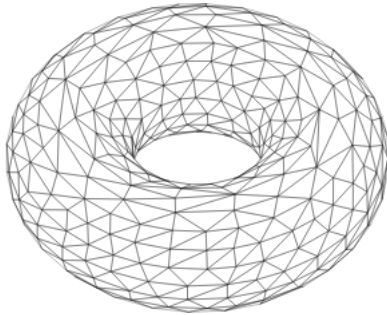
@jbetzend



Mein Thema

*Wie kann ich einem Computer beibringen,
mit mathematischen Beweisen umzugehen?*





Mathematik ist eine Wissenschaft
fast wie jede andere. . .

*Math is the reverse of comedy. The anti-joke.
We'll tell you the punchline first, then laboriously explain
to you why it was the right punchline.*

Joseph Maher, College of Staten Island

GIVEN THE PACE OF
TECHNOLOGY, I PROPOSE
WE LEAVE MATH TO THE
MACHINES AND GO PLAY
OUTSIDE.



David Hilbert

Mathematiker (1862-1943) aus Königsberg, bekannt für seine Grundlagenforschung und Problemsammlungen.

Hilberts Traum:

Ein Computer, dem ich eine beliebige mathematische Aussage reichen kann und der mir sagt, ob sie stimmt.



Das ist leicht zu schaffen, wenn wir das *Halteproblem* lösen! (z.B. durch Auflistung aller Folgerungen aus den Axiomen)

Halteproblem:

Gegeben nur den Quellcode und den Input eines Programms, sage vorher ob dieses Programm jemals „fertig wird“ oder endlos weiter läuft.

```

extract_num=
{dest, and, inc} {destination, source} int
/*destination: unsigned char *source; extract_num=
low {destination, source; source += 2;} #ifdef EXTRACT_MAC-
ROS #define EXTRACT_NUMBER_AND_INC #define EXTRACT_NUM-
BER_AND_INC #else #define EXTRACT_NUMBER_AND_INC #endif /*
not EXTRACT_MACROS */ #endif /* DEBUG */ #if DEBUG is defined, Regex prints
many voluminous messages about what it is doing if the variable 'debug' is non-zero. If
linked with the main program in 'imgexec', you can enter patterns and strings interactively.
And if linked with the main program in 'main.c', and the other test files, you can run the al-
ready-written tests. */ #ifdef DEBUG /* We use standard I/O for debugging. */ #include <stdio.h>
/* It is useful to test things that "must" be true when debugging. */ #include <assert.h> static int
debug = 0; #define DEBUG_STATEMENT(x) #define DEBUG_PRINT(x) if (debug) print (x) #define
DEBUG_PRINT2(x1, x2) if (debug) printf (x1, x2) #define DEBUG_PRINT3(x1, x2, x3) if (debug) printf
(x1, x2, x3) #define DEBUG_PRINT4(x1, x2, x3, x4) if (debug) printf (x1, x2, x3, x4) #define DE-
BUG_PRINT_COMPILED_PATTERN(p, x, y) if (debug) print_partial_compiled_pattern (x, y) #define DE-
BUG_PRINT_DOUBLE_STRING(s1, s2, t1, t2) if (debug) print_double_string (s1, s2, t1, t2)
extern void printchar(); /* Print the fastmap in human-readable form. */ void print_fastmap (fastmap)
char *fastmap; { unsigned was_a_range = 0; unsigned i = 0; while (i < (1 << BYTWIDTH)) { if (fastmap[i++]
) { was_a_range = 0; printchar (0 - 1); while (i < (1 << BYTWIDTH)) && fastmap[i] { was_a_range = 1; i++; if
!was_a_range) { print (" "); printchar (0 - 1); } } } } } /* Print a compiled pattern string in hu-
man-readable form, starting at the START pointer into it and ending just before the pointer END. */ void
print_partial_compiled_pattern (start, end; unsigned char *start; unsigned char *end; int more; more; un-
signed char *p = start; unsigned char *pend = end; if (start == NULL) { print ("null"); return; } /* Loop over
pattern commands. */ while (p < pend) { switch line_opcode (p++) { case no_op: print ("no_op");
break; case exact: more = "p++"; print ("%exact/%d", more); do { print (" "); printchar ("p++"); }
while (!more); break; case start_memory: more = "p++"; print ("start_memory/%d", more);
"p++"; break; case stop_memory: more = "p++"; print ("stop_memory/%d", more); "p++";
break; case duplicate: print ("%duplicate/%d", "p++"); break; case append: print ("%append");
break; case charset: case charset_not: { register int c; print ("%charset/%d (%c, opcode: %d %p-
1) == charset, 'not 1', 'not': "); assert (p + "p < pend"); for (c = 0; c < "p < c + 1) { unsigned b;
assigned char map_byte = (p[1 + c]); printchar (" "); for (bit = 0; bit < BYTWIDTH; bit++) if
(map_byte & (1 << bit)) printchar (" "); p += 1 + "p; break; case beg-
line: print ("beginline"); break; case endline: print ("endline"); break; case on_failure_-
jump: extract_number_and_inc (&dest, &p); print ("on_failure_jump/%d", more);
break; case on_failure_keep_string: jump: extract_number_and_inc (&dest, &p); print
("on_failure_keep_string_jump/%d", more); break; case dummy_failure_jump:
extract_number_and_inc (&dest, &p); print ("dummy_failure_jump/%d", more);
break; case path_dummy_failure: print ("path_dummy_failure"); break; case map-
be_pop_jump: extract_number_and_inc (&dest, &p); print
("maybe_pop_jump/%d", more); break; case pop_failure_-
jump: extract_number_and_inc (&dest, &p); print ("pop_-
failure_jump/%d", more); break; case jump_post_ift-
extract_number_and_inc (&dest, &p); print ("");

```

Manchmal ist es sehr leicht zu sehen, ob ein Programm jemals halten wird oder nicht:

```
-- Dieses Programm hält quasi sofort  
main :: IO ()  
main = print $ plus (3,4)  
  where  
    plus :: (Int, Int) -> Int  
    plus (x,y) = x + y
```


Manchmal ist es sehr leicht zu sehen, ob ein Programm jemals halten wird oder nicht:

```
-- Dieses Programm hält quasi sofort  
main :: IO ()  
main = print $ plus (3,4)  
  where  
    plus :: (Int, Int) -> Int  
    plus (x,y) = x + y
```

```
-- Dieses Programm läuft "für immer"  
main :: IO ()  
main = forever $ print "lol, infinite loop!"
```

Manchmal ist es aber auch nahezu unmöglich!

```
-- Nobody knows if this ever halts...
main :: IO ()
main = do let results = filter isPerfect [1,3..]
          case results of
            [] -> print " No odd perfect numbers!"
            _  -> print "Yes odd perfect numbers!"

divisors :: Int -> [Int]
divisors n = filter (\x -> n `rem` x == 0) [1..n]

isPerfect :: Int -> Bool
isPerfect n = (sum . divisors) n == n + n
```

Alan Turing



Mathematiker (1912-1954) aus London. Half in Bletchley Park, den dt. *Enigma*-Code zu lösen.

Bewies in seiner Doktorarbeit, dass das Halteproblem *nicht lösbar sein kann* (zumindest im allgemeinen Fall)!

Alan Turing



Mathematiker (1912-1954) aus London. Half in Bletchley Park, den dt. *Enigma*-Code zu lösen.

Bewies in seiner Doktorarbeit, dass das Halteproblem *nicht lösbar sein kann* (zumindest im allgemeinen Fall)!

Aber nehmen wir mal an, es *wäre lösbar*...

Ein magisches Halte-Orakel (1)



Angenommen wir haben ein magisches Orakel, was uns für Quellcode und Input immer sagt, ob ein Programm hält oder nicht.

Ein magisches Halte-Orakel (2)

Quellcode ->

Input ->



-> Ergebnis
(Hält oder
hält nicht)

Angenommen wir haben ein magisches Orakel, was uns für Quellcode und Input immer sagt, ob ein Programm hält oder nicht.

Ein magisches Halte-Orakel (3)



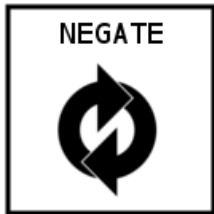
Angenommen wir haben ein magisches Orakel, was uns für Quellcode und Input immer sagt, ob ein Programm hält oder nicht.

Ein magisches Halte-Orakel (4)



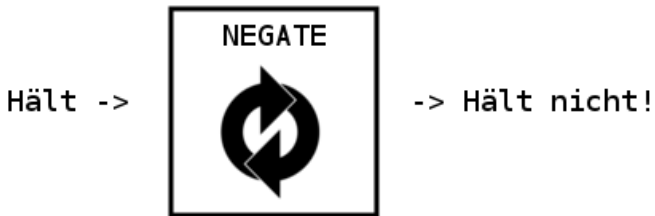
Angenommen wir haben ein magisches Orakel, was uns für Quellcode und Input immer sagt, ob ein Programm hält oder nicht.

Negierer (1)



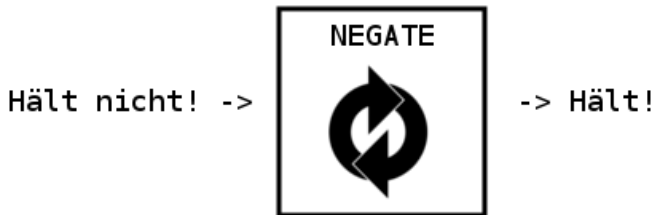
Dazu kommt ein Negierer, der ein Haltergebnis umdreht!

Negierer (2)



Dazu kommt ein Negierer, der ein Haltergebnis umdreht!

Negierer (3)



Dazu kommt ein Negierer, der ein Haltergebnis umdreht!

Duplizierer (1)



Wir haben auch einen Duplizierer, der seine Eingabe verdoppelt!

Duplizierer (2)



Wir haben auch einen Duplizierer, der seine Eingabe verdoppelt!

Duplizierer (3)



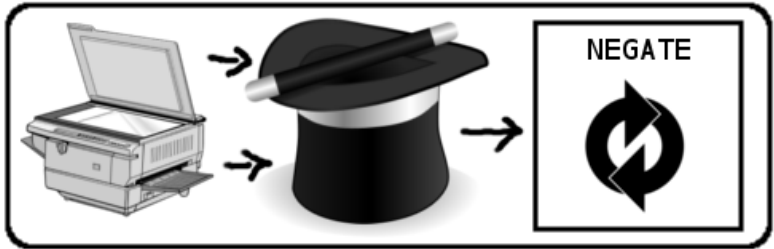
Wir haben auch einen Duplizierer, der seine Eingabe verdoppelt!

Unite and conquer! (1)



Manche ahnen es vielleicht schon! Alles das können wir jetzt zusammen stecken und hintereinander schalten!

Unite and conquer! (2)



Insbesondere können wir diese Aneinanderreihung als *eine Maschine* betrachten. Nennen wir sie X.

Unite and conquer! (3)

A large, bold, black 'X' is centered within a teal-colored rounded rectangle with a thin black border. The rectangle has rounded corners and is positioned in the upper half of the slide.

Insbesondere können wir diese Aneinanderreihung als *eine Maschine* betrachten. Nennen wir sie X.

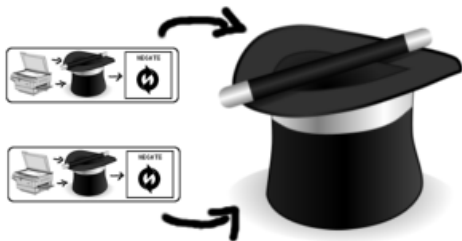
Going Meta! (1)



Jetzt nur keine Panik bekommen!

Aber was passiert, wenn wir den Quellcode von X an X geben?

Going Meta! (2)



Der interessante Teil der Frage ist, was das Orakel ausspuckt.
Hier gibt es aber überhaupt nur zwei Möglichkeiten!

Fall 1: Orakel sagt "Hält!"



Entweder das Orakel sagt das X mit X als Inpt hält. Das geht aber nicht, weil dann der Negierer dafür sorgt, dass es *nicht* hält.

Fall 2: Orakel sagt "Hält nicht!"



Oder das Orakel sagt X mit X als Inpt hält *nicht*! Das bringt aber das gleiche Problem: dank des Negierers hält X eben doch!

Widerspruch



Was bleibt uns von unserer erdachten magischen Maschine?
Jeder mögliche Weg führt zum Widerspruch!

Widerspruch



Was bleibt uns von unserer erdachten magischen Maschine?
Jeder mögliche Weg führt zum Widerspruch!

Damit ist bewiesen: so ein Orakel *kann* es nicht geben!
Das Halteproblem ist (im Allgemeinen) unlösbar!

Widerspruch



Was bleibt uns von unserer erdachten magischen Maschine?
Jeder mögliche Weg führt zum Widerspruch!

Damit ist bewiesen: so ein Orakel *kann* es nicht geben!
Das Halteproblem ist (im Allgemeinen) unlösbar!

Schlusswort

Was haben wir heute gelernt?



Schlusswort

Was haben wir heute gelernt?

- Die Welt ist groß und ungewiss!



Schlusswort

Was haben wir heute gelernt?

- Die Welt ist groß und ungewiss!
- Mathe macht Spaß!



Schlusswort



Was haben wir heute gelernt?

- Die Welt ist groß und ungewiss!
- Mathe macht Spaß!
- Wir können jetzt das Halteproblem erklären und wissen, dass es keine allgemeine Lösung gibt.

Schlusswort



Was haben wir heute gelernt?

- Die Welt ist groß und ungewiss!
- Mathe macht Spaß!
- Wir können jetzt das Halteproblem erklären und wissen, dass es keine allgemeine Lösung gibt.

*Bitte bleiben
Sie neugierig!*