

SAT is Not Solvable in Constant Time

Daniel Shea

April 4, 2026

Abstract

This entry establishes that the Boolean Satisfiability Problem (SAT) cannot be decided by a deterministic Turing machine in constant time ($O(1)$).

The core of the argument rests on an indistinguishability invariant: a Turing machine bounded by constant time C can only ever read the first $C + 1$ cells of its input tape. Consequently, any language decided in constant time must be a prefix language. We then show that SAT is not uniquely determined by any finite prefix, yielding the impossibility result.

Contents

1	Constant-Time Lower Bounds	1
1.1	Head-Movement Bounds for Turing Machines	2
1.2	Configuration Indistinguishability	3
1.3	Constant-Time Dependence on Prefix	5
1.4	Main Theorem: $\text{SAT} \notin \text{DTIME}(1)$	6

theory *SAT-Not-Const-Time*

imports

Main

Cook-Levin.Reduction-TM

begin

1 Constant-Time Lower Bounds

This theory establishes that the Boolean Satisfiability Problem (SAT) cannot be decided by a deterministic Turing machine in constant time ($O(1)$).

The core of the argument rests on an indistinguishability invariant: a Turing machine bounded by constant time C can only ever read the first $C + 1$ cells of its input tape. Consequently, any language decided in constant time must be a prefix language. We then show that SAT is not uniquely determined by any finite prefix, yielding the impossibility result.

Configuration records and the step function are inherited from the Cook-Levin formalization [1].

1.1 Head-Movement Bounds for Turing Machines

We first establish that a Turing machine's head can move at most one cell per execution step. Configuration records and the step function are inherited from *Reduction-TM*. We isolate the position of the input-tape head (tape 0).

definition *hd-span* :: *config* \Rightarrow *nat* **where**
hd-span *cfg* \equiv *cfg* <#> 0

lemma *le-imp-le-Suc*: $n \leq m \implies n \leq \text{Suc } m$
 <proof>

lemma *contents-nil* [*simp*]:
 $[\] = (\lambda i :: \text{nat}. \text{if } i = 0 \text{ then } 1 \text{ else } 0)$
 <proof>

lemma *start-config-tape1* [*simp*]:
assumes $2 \leq k$
shows $(\text{start-config } k \text{ } xs \text{ } <:> 1) = [\]$
 <proof>

lemma *start-config-string-tape1* [*simp*]:
assumes $2 \leq k$
shows $\text{snd } (\text{start-config-string } k \text{ } w) :: 1 = [\]$
 <proof>

lemma *config-tape-snd* [*simp*]:
 $(\text{snd } \text{cfg} :: j) = (\text{cfg } <:> j)$
 <proof>

lemma *fst-start-config-string* [*simp*]:
 $\text{fst } (\text{start-config-string } k \text{ } w) = 0$
 <proof>

lemma *act-move-bound*:
fixes *tp tp' :: tape* **and** *a :: action*
assumes $tp' = \text{act } a \text{ } tp$
shows $\text{hd-span } (q, [tp'] @ tps) \leq \text{Suc } (\text{hd-span } (q, [tp] @ tps))$
 <proof>

lemma *nth-map2*:
assumes $\text{length } xs = \text{length } ys \text{ } i < \text{length } xs$
shows $(\text{map2 } f \text{ } xs \text{ } ys) ! i = f (xs ! i) (ys ! i)$
 <proof>

lemma *map2-act-nth0*:
assumes $length\ as = length\ ts\ ts \neq []$
shows $map2\ act\ as\ ts\ !\ 0 = act\ (as\ !\ 0)\ (ts\ !\ 0)$
 $\langle proof \rangle$

lemma *map2-act-nth0-rewrite*:
assumes $as \neq []\ ts \neq []$
shows $map2\ act\ as\ ts\ !\ 0 = act\ (as\ !\ 0)\ (ts\ !\ 0)$
 $\langle proof \rangle$

lemma *tapes-pos-by-no-unfold* [*simp*]:
 $(tps :: tape\ list) : \# : j = snd\ (tps\ !\ j)$
 $\langle proof \rangle$

The head position on the input tape is bounded by the number of steps taken.

lemma *sem-head-move-bound*:
fixes $cmd :: command$ **and** $cfg\ cfg' :: config$
assumes $pc : proper-command\ ||cfg||\ cmd$
and $step : cfg' = sem\ cmd\ cfg$
shows $hd-span\ cfg' \leq Suc\ (hd-span\ cfg)$
 $\langle proof \rangle$

lemma *exe-head-move-bound*:
fixes $cfg\ cfg' :: config$
assumes $pm : proper-machine\ ||cfg||\ M$
and $step : cfg' = exe\ M\ cfg$
shows $hd-span\ cfg' \leq Suc\ (hd-span\ cfg)$
 $\langle proof \rangle$

lemma *execute-head-pos-le-time*:
fixes $M :: machine$ **and** $cfg\ cfg' :: config$
assumes $pm : proper-machine\ ||cfg||\ M$
and $step : cfg' = execute\ M\ cfg\ t$
shows $hd-span\ cfg' \leq hd-span\ cfg + t$
 $\langle proof \rangle$

1.2 Configuration Indistinguishability

Two machine configurations are completely indistinguishable to the execution semantics for at least one step if their control states match, their non-input tapes are identical, and their input tapes agree up to the strict bounds of where the read head can currently be located.

definition $indist :: nat \Rightarrow config \Rightarrow config \Rightarrow bool$ **where**
 $indist\ C\ cfg1\ cfg2 \equiv$
 $||cfg1|| = ||cfg2|| \wedge$
 $fst\ cfg1 = fst\ cfg2 \wedge$
 $(\forall j > 0. snd\ cfg1\ !\ j = snd\ cfg2\ !\ j) \wedge$

$$(\forall i \leq C. (cfg1 \langle :> 0 \rangle i = (cfg2 \langle :> 0 \rangle i) \wedge \\ hd\text{-span } cfg1 = hd\text{-span } cfg2 \wedge hd\text{-span } cfg1 \leq C)$$

lemma *start-config-string-conv* [*simp*]:
start-config-string k $w = \text{start-config } k$ (*string-to-symbols* w)
 ⟨*proof*⟩

lemma *big-oh-constE*:
assumes *big-oh* T ($\lambda n. \text{Suc } 0$)
obtains C **where** $\forall n. T\ n \leq C$
 ⟨*proof*⟩

lemma *contents-eq-on-prefix*:
assumes *EQ*: *take* (*Suc* C) $w = \text{take } (Suc\ C)$ v
shows $\forall i \leq Suc\ C.$
 $\lfloor \text{map } (\lambda b. \text{if } b \text{ then } 3 \text{ else } 2) w \rfloor i =$
 $\lfloor \text{map } (\lambda b. \text{if } b \text{ then } 3 \text{ else } 2) v \rfloor i$
 ⟨*proof*⟩

lemma *DTIME1-const-time*:
assumes $L \in \text{DTIME } (\lambda n. 1)$
obtains $k\ G\ M\ C$ **where**
turing-machine $k\ G\ M$
computes-in-time $k\ M$ (*characteristic* L) ($\lambda-. C$)
 ⟨*proof*⟩

lemma *start-input-prefix-eq*:
assumes *take* (*Suc* C) $w = \text{take } (Suc\ C)$ v
and $i \leq C$
shows (*start-config-string* $k\ w \langle :> 0 \rangle i =$
start-config-string $k\ v \langle :> 0 \rangle i$)
 ⟨*proof*⟩

lemma *indist-start*:
assumes *EQ*: *take* (*Suc* C) $w = \text{take } (Suc\ C)$ v
and $k: 2 \leq k$
shows *indist* C (*start-config-string* $k\ w$) (*start-config-string* $k\ v$)
 ⟨*proof*⟩

lemma *config-read-eq-indist*:
assumes *indist* $C\ cfg1\ cfg2$
shows *config-read* $cfg1 = \text{config-read } cfg2$
 ⟨*proof*⟩

lemma *nth-map-act-zip*:
assumes $j: j < \text{length } as$ **and** $\text{len}: \text{length } as = \text{length } tps$
shows *map* ($\lambda(a, tp). \text{act } a\ tp$) (*zip* $as\ tps$) ! $j = \text{act } (as\ !\ j)$ ($tps\ !\ j$)
 ⟨*proof*⟩

lemma *nth-Nil* [simp]: ($[] :: 'a \text{ list}$) ! $n = \text{undefined } n$
 <proof>

lemma *nth-overflow* [simp]:
 $\text{length } xs \leq n \implies xs ! n = \text{undefined } (n - \text{length } xs)$
 <proof>

lemma *act-prefix-eq*:
assumes *cont*: $\forall i \leq C. \text{fst } (tp1 :: \text{tape}) i = \text{fst } tp2 i$
and *head*: $\text{snd } tp1 = \text{snd } tp2$
shows $\forall i \leq C. \text{fst } (\text{act } a \text{ } tp1) i = \text{fst } (\text{act } a \text{ } tp2) i$
 <proof>

lemma *act-head-eq*:
assumes $\text{snd } (tp1 :: \text{tape}) = \text{snd } tp2$
shows $\text{snd } (\text{act } a \text{ } tp1) = \text{snd } (\text{act } a \text{ } tp2)$
 <proof>

lemma *indist-step*:
assumes *inv* : $\text{indist } C \text{ } cfg1 \text{ } cfg2$
and *pc* : $\text{proper-command } ||cfg1|| \text{ } cmd$
and *bounds*: $\text{hd-span } cfg1 < C$
and *s1* : $cfg1' = \text{sem } cmd \text{ } cfg1$
and *s2* : $cfg2' = \text{sem } cmd \text{ } cfg2$
shows $\text{indist } C \text{ } cfg1' \text{ } cfg2'$
 <proof>

lemma *indist-execute*:
assumes *pm*: $\text{proper-machine } ||cfg|| \text{ } M$
and *inv0*: $\text{indist } C \text{ } cfg1 \text{ } cfg2$
and *len*: $||cfg1|| = ||cfg||$
and *hd-bound*: $\text{hd-span } cfg1 + t \leq C$
shows $\text{indist } C \text{ } (\text{execute } M \text{ } cfg1 \text{ } t) \text{ } (\text{execute } M \text{ } cfg2 \text{ } t)$
 <proof>

1.3 Constant-Time Dependence on Prefix

By applying the *indist_execute* invariant up to step C , we can easily show that two executions starting with the same prefix must deposit the exact same output onto tape 1.

lemma *exec-same-output-C*:
fixes $k \ C \ t :: \text{nat}$ **and** $w \ v :: \text{string}$ **and** $G :: \text{nat}$ **and** $M :: \text{machine}$
defines $cfgw \equiv \text{start-config-string } k \ w$
and $cfgv \equiv \text{start-config-string } k \ v$
assumes *TM* : $\text{turing-machine } k \ G \ M$
and *EQ* : $\text{take } (\text{Suc } C) \ w = \text{take } (\text{Suc } C) \ v$
and *LE* : $t \leq C$
shows $(\text{execute } M \text{ } cfgw \ t <:> 1) = (\text{execute } M \text{ } cfgv \ t <:> 1)$
 <proof>

lemma *string-to-contents-inj*:
assumes *string-to-contents* $xs = \text{string-to-contents } ys$
shows $xs = ys$
 $\langle \text{proof} \rangle$

lemma *constant-time-depends-on-prefix*:
fixes $f :: \text{string} \Rightarrow \text{string}$
assumes $TM: \text{turing-machine } k \ G \ M$
and $CT: \text{computes-in-time } k \ M \ f \ (\lambda \cdot. \ C)$
and $EQ: \text{take } (Suc \ C) \ w = \text{take } (Suc \ C) \ v$
shows $f \ w = f \ v$
 $\langle \text{proof} \rangle$

lemma *DTIME1-depends-on-prefix*:
fixes $L :: \text{language}$
assumes $L \in \text{DTIME } (\lambda n. \ 1)$
obtains $C \ H$ **where** $\forall w. w \in L \longleftrightarrow \text{take } (Suc \ C) \ w \in H$
 $\langle \text{proof} \rangle$

1.4 Main Theorem: $SAT \notin \text{DTIME}(1)$

The core diagonalization-style argument works by constructing two strings: v , which is properly formatted but trivially unsatisfiable, and w , which shares the same $O(1)$ prefix with v but is engineered via an odd length to be syntactically invalid (and thus conditionally belongs to SAT due to the library's inverted validity mapping for the default case). The hypothesized $O(1)$ decider contradicts itself on this shared prefix.

lemma *SAT-not-prefix-language*:
shows $\neg (\exists C \ H. \forall w. w \in \text{SAT} \longleftrightarrow \text{take } (Suc \ C) \ w \in H)$
 $\langle \text{proof} \rangle$

theorem *SAT-not-const-time*:
shows $\text{SAT} \notin \text{DTIME } (\lambda n. \ 1)$
 $\langle \text{proof} \rangle$

end

References

- [1] F. J. Balbach. The cook-levin theorem. *Archive of Formal Proofs*, January 2023. https://isa-afp.org/entries/Cook_Levin.html, Formal proof development.