SAT is Not Solvable in Constant Time

Daniel Shea

November 15, 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 5
1.3 Constant-Time Dependence on Prefix 14
1.4 Main Theorem: SAT $\notin$ DTIME(1) 17

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* ⇒ *nat* **where**
  hd-span cfg ≡ *cfg* <#> *0*

lemma *le-imp-le-Suc*: *n* ≤ *m* ⟹ *n* ≤ *Suc m*
  by *simp*

lemma *contents-nil* [*simp*]:
  ⌊[]⌋ = (λ*i*::*nat*. *if i* = *0 then 1 else 0*)
  unfolding *contents-def*
  by *auto*

lemma *start-config-tape1* [*simp*]:
  assumes *2* ≤ *k*
  shows (*start-config k xs* <:> *1*) = ⌊[]⌋
proof −
  obtain *k′* **where** *k′*: *k* = *Suc* (*Suc k′*)
    using *assms nat-le-iff-add* **by** *auto*
  show *?thesis*
    unfolding *start-config-def k′*
    by *simp*
qed

lemma *start-config-string-tape1*[*simp*]:
  assumes *2* ≤ *k*
  shows *snd* (*start-config-string k w*) ::: *1* = ⌊[]⌋
  using *assms start-config-tape1* **by** *auto*

lemma *config-tape-snd*[*simp*]:
  (*snd cfg* ::: *j*) = (*cfg* <:> *j*)
  by (*cases cfg*) *simp*

lemma *fst-start-config-string* [*simp*]:
  fst (*start-config-string k w*) = *0*
  unfolding *start-config-def* **by** *simp*

lemma *act-move-bound*:
  fixes *tp tp′* :: *tape* **and** *a* :: *action*
  assumes *tp′* = *act a tp*
  shows *hd-span* (*q*, [*tp′*] @ *tps*) ≤ *Suc* (*hd-span* (*q*, [*tp*] @ *tps*))

unfolding *hd-span-def assms* **by** (*cases a*) (*auto split*: *direction.split*)

lemma *nth-map2*:
assumes *length xs* = *length ys* *i* < *length xs*
shows (*map2 f xs ys*) ! *i* = *f* (*xs* ! *i*) (*ys* ! *i*)
using *assms*
by (*induction xs ys rule*: *list-induct2*) (*auto simp add*: *nth-Cons'*)

lemma *map2-act-nth0*:
assumes *length as* = *length ts* *ts* ≠ []
shows *map2 act as ts* ! *0* = *act* (*as* ! *0*) (*ts* ! *0*)
using *nth-map2*[*of as ts 0 act*] *assms* **by** *simp*

lemma *map2-act-nth0-rewrite*:
assumes *as* ≠ [] *ts* ≠ []
shows *map2 act as ts* ! *0* = *act* (*as* ! *0*) (*ts* ! *0*)
using *assms*(*1*,*2*) **by** *force*

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

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'* ≤ *Suc* (*hd-span cfg*)
proof −
obtain *q tps* **where** *cfg*: *cfg* = (*q*, *tps*) **by** (*cases cfg*) *auto*
then have *len-cfg*: ||*cfg*|| = *length tps* **by** *simp*

have *len-read*: *length* (*config-read cfg*) = ||*cfg*||
using *cfg read-length len-cfg* **by** *simp*

obtain *st acts* **where** *sas*: *cmd* (*config-read cfg*) = (*st*, *acts*)
by (*cases cmd* (*config-read cfg*)) *blast*
from *step cfg sas* **have** *cfg'*:
cfg' = (*st*, *map* (λ(*a*,*tp*). *act a tp*) (*zip acts tps*))
by (*simp add*: *sem-def*)

have *acts-len*: *length acts* = *length tps*
using *proper-command-length*[*OF pc len-read*] *sas len-cfg*
using *len-read* **by** *fastforce*

show *?thesis*
proof (*cases tps*)
case *Nil*

```
    with cfg cfg' show ?thesis
      by (metis Suc-n-not-le-n hd-span-def linorder-le-cases
          list.map-disc-iff snd-eqD zip-eq-Nil-iff)
  next
    case (Cons tp tps')
    then obtain a as where acts: acts = a # as
      by (metis Suc-length-conv acts-len)

    define tps0 where tps0 ≡ tp # tps'

    from cfg' Cons acts tps0-def have cfg'1:
      cfg' = (st, map2 act (a # as) tps0) by simp

    have hd-new: hd-span cfg' = snd (act a tp)
      unfolding cfg'1 hd-span-def
      using Cons map2-act-nth0-rewrite[of a # as tps0] by (simp add:tps0-def)

    have hd-old: hd-span cfg = snd tp
      unfolding cfg hd-span-def Cons by simp

    have hd-span (st, (act a tp) # tps') ≤ Suc (hd-span (st, tp # tps'))
      using act-move-bound by auto
    then have snd (act a tp) ≤ Suc (snd tp)
      by (simp add: hd-span-def)

    with hd-new hd-old show ?thesis by simp
  qed
qed

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' ≤ Suc (hd-span cfg)
proof −
  obtain q tps where cfg [simp]: cfg = (q,tps) by (cases cfg)
  show ?thesis
  proof (cases q < length M)
    case False
    hence cfg' = cfg using step exe-def by simp
    thus ?thesis by simp
  next
    case True
    define cmd where cmd = M ! q
    from step True cmd-def have cfg': cfg' = sem cmd cfg
      by (simp add: exe-def)
    have proper-cmd: proper-command ||cfg|| cmd
      using pm True cmd-def by simp
    from sem-head-move-bound[OF proper-cmd cfg'] show ?thesis .
```

```
  qed
qed

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' ≤ hd-span cfg + t
using step pm
proof (induction t arbitrary: cfg cfg')
  case 0
  then show ?case by simp
next
  case (Suc t cfg cfg')
  define cfg1 where cfg1 = execute M cfg t
  have IH: hd-span cfg1 ≤ hd-span cfg + t
    using Suc.IH Suc.prems cfg1-def by simp

  have step1: cfg' = exe M cfg1
    by (simp add: Suc.prems(1) cfg1-def)

  have tapes-eq: ||cfg1|| = ||cfg||
    using Suc.prems(2) cfg1-def execute-num-tapes-proper by auto
  have pm1: proper-machine ||cfg1|| M
    by (simp add: Suc.prems(2) tapes-eq)

  have hd-span cfg' ≤ Suc (hd-span cfg1)
    using exe-head-move-bound[OF pm1 step1] .
  also have ... ≤ Suc (hd-span cfg + t)
    using IH by simp
  finally show ?case by simp
qed
```

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 ⇒ config ⇒ config ⇒ bool where
  indist C cfg1 cfg2 ≡
    ||cfg1|| = ||cfg2|| ∧
    fst cfg1 = fst cfg2 ∧
    (∀ j>0. snd cfg1 ! j = snd cfg2 ! j) ∧
    (∀ i≤C. (cfg1 <:> 0) i = (cfg2 <:> 0) i) ∧
    hd-span cfg1 = hd-span cfg2 ∧ hd-span cfg1 ≤ C

lemma start-config-string-conv [simp]:
  start-config-string k w = start-config k (string-to-symbols w)
```

```
  by simp

lemma big-oh-constE:
  assumes big-oh T (λn. Suc 0)
  obtains C where ∀ n. T n ≤ C
proof −
  obtain c m where h: ∀ n>m. T n ≤ c
    using assms unfolding big-oh-def by auto
  have ∀ n≤m. T n ≤ Max (insert c {T n | n. n ≤ m})
    by (auto intro!: Max-ge)
  moreover have ∀ n>m. T n ≤ Max (insert c {T n | n. n ≤ m})
    using h by (simp add: Max-ge-iff)
  ultimately have ∀ n. T n ≤ Max (insert c {T n | n. n ≤ m})
    by (meson linorder-le-less-linear)
  thus ?thesis ..
qed

lemma contents-eq-on-prefix:
  assumes EQ: take (Suc C) w = take (Suc C) v
  shows   ∀ i≤Suc C.
            ⌊map (λb. if b then 3 else 2) w⌋ i =
            ⌊map (λb. if b then 3 else 2) v⌋ i
proof (intro allI impI)
  fix i :: nat
  assume bound: i ≤ Suc C
  let ?f = λb::bool. if b then 3 else 2

  show ⌊map ?f w⌋ i = ⌊map ?f v⌋ i
  proof (cases i)
    case 0
    thus ?thesis by (simp add: contents-def)
  next
    case (Suc j)
    have j-lt: j < Suc C using bound Suc by simp

    note take-map = arg-cong[OF EQ, of map ?f]
    have nth-eq:
      map ?f w ! j = map ?f v ! j
      using j-lt take-map
      by (metis (lifting) ext List.take-map nth-take)

    show ?thesis
    proof (cases i ≤ length w)
      case True
      hence j < length w using Suc by simp
      moreover have j < length v
      proof −
        have length (take (Suc C) w) = length (take (Suc C) v)
          using EQ by simp
```

```
      hence min (Suc C) (length w) = min (Suc C) (length v)
        by (simp add: min.commute)
      with ‹j < length w› j-lt show ?thesis
        by (cases length w < Suc C; cases length v < Suc C) simp-all
    qed
    ultimately show ?thesis
      using nth-eq True
      by (simp add: Suc)
  next
    case False
    then have len-w: length w < i
      by (simp add: Suc)
    have len-v: length v < i
    proof (rule ccontr)
      assume ¬ length v < i
      then have i-le: i ≤ length v by simp
      have take-i-eq: take i w = take i v
        using EQ bound
        by (metis min.absorb1 take-take)
      have take i w = w
        using len-w by simp
      hence length w = length (take i v)
        using take-i-eq by simp
      moreover have length (take i v) = i
        using i-le by simp
      ultimately have length w = i by simp
      with len-w show False by simp
    qed
    from len-w len-v
    show ?thesis by simp
  qed
qed

lemma DTIME1-const-time:
  assumes L ∈ DTIME (λn.1)
  obtains k G M C where
      turing-machine k G M
      computes-in-time k M (characteristic L) (λ-. C)
proof −
  obtain k G M T  where
  tm: turing-machine k G M
      big-oh T (λn.1)
      computes-in-time k M (characteristic L) T
    using assms unfolding DTIME-def by blast

  obtain C where C: ∀ n. T n ≤ C
    using tm(2) using big-oh-constE by auto
```

```
  have computes-in-time k M (characteristic L) (λ-. C)
    using tm(3) C
    unfolding computes-in-time-def
    by (meson linorder-le-less-linear order-less-imp-not-eq2
        order-less-le-trans)

  thus ?thesis
    using that tm(1) by auto
qed

lemma start-input-prefix-eq:
  assumes take (Suc C) w = take (Suc C) v
      and i ≤ C
  shows   (start-config-string k w <:> 0) i =
          (start-config-string k v <:> 0) i
proof −
  have contents-eq:
    ⌊string-to-symbols w⌋ i = ⌊string-to-symbols v⌋ i
  proof −
    have ⌊map (λb. if b then 3 else 2) w⌋ i =
         ⌊map (λb. if b then 3 else 2) v⌋ i
      using assms(1,2) contents-eq-on-prefix le-imp-le-Suc
      by blast
    thus ?thesis
      by blast
  qed
  have lhs: (start-config-string k w <:> 0) i = ⌊string-to-symbols w⌋ i
    by (simp add: start-config-def)
  have rhs: (start-config-string k v <:> 0) i = ⌊string-to-symbols v⌋ i
    by (simp add: start-config-def)
  from contents-eq lhs rhs show ?thesis
    by presburger
qed

lemma indist-start:
  assumes EQ: take (Suc C) w = take (Suc C) v
    and k:  2 ≤ k
  shows     indist C (start-config-string k w) (start-config-string k v)
  unfolding indist-def
  using EQ start-input-prefix-eq[OF EQ] k
  by (auto simp: hd-span-def start-config-def)

lemma config-read-eq-indist:
  assumes indist C cfg1 cfg2
  shows   config-read cfg1 = config-read cfg2
proof −
  obtain q1 tps1 where cfg1: cfg1 = (q1,tps1) by (cases cfg1)
  obtain q2 tps2 where cfg2: cfg2 = (q2,tps2) by (cases cfg2)
  from assms have len: length tps1 = length tps2
```

```
    unfolding indist-def cfg1 cfg2 by simp
  have read-eq: tape-read (tps1 ! j) = tape-read (tps2 ! j)
    if j < length tps1 for j
  proof (cases j)
    case 0
    then show ?thesis
      using assms unfolding indist-def cfg1 cfg2
      by (metis hd-span-def snd-conv)
  next
    case (Suc i)
    with that have j > 0 by simp
    moreover from assms have snd cfg1 ! j = snd cfg2 ! j
      unfolding indist-def cfg1 cfg2 using ‹j>0› by simp
    ultimately show ?thesis
      using ‹j > 0›
      by (metis cfg1 cfg2 fst-conv snd-swap swap-simp)
  qed
  show ?thesis
    unfolding cfg1 cfg2 read-def
    by (metis (mono-tags, lifting) len map-equality-iff read-eq
        snd-conv)
qed

lemma nth-map-act-zip:
  assumes j: j < length as and len: length as = length tps
  shows map (λ(a,tp). act a tp) (zip as tps) ! j = act (as ! j) (tps ! j)
  using j len by auto

lemma nth-Nil [simp]: ([] :: 'a list) ! n = undefined n
  by (simp add: nth-def hd-def)

lemma nth-overflow [simp]:
  length xs ≤ n ⟹ xs ! n = undefined (n − length xs)
  by (induct xs arbitrary: n) (auto split: nat.split)

lemma act-prefix-eq:
  assumes cont: ∀ i≤C. fst (tp1 :: tape) i = fst tp2 i
     and head: snd tp1 = snd tp2
  shows  ∀ i≤C. fst (act a tp1) i = fst (act a tp2) i
  using cont head
  by (cases a) (auto split: direction.split)

lemma act-head-eq:
  assumes snd (tp1:: tape) = snd tp2
  shows  snd (act a tp1) = snd (act a tp2)
  using assms
  by (metis act split-pairs)

lemma indist-step:
```

```
  assumes inv : indist C cfg1 cfg2
      and pc  : proper-command ||cfg1|| cmd
      and bounds: hd-span cfg1 < C
      and s1  : cfg1' = sem cmd cfg1
      and s2  : cfg2' = sem cmd cfg2
    shows       indist C cfg1' cfg2'
proof −
  have rs-eq: config-read cfg1 = config-read cfg2
    using inv config-read-eq-indist by blast

  have st-eq: fst (sem cmd cfg1) = fst (sem cmd cfg2)
    unfolding sem-def rs-eq
    by (simp add: case-prod-beta)

  obtain st acts where acts: cmd (config-read cfg1) = (st, acts)
    by (cases cmd (config-read cfg1)) blast
  hence cmd2: cmd (config-read cfg2) = (st, acts)
    by (simp add: rs-eq)

  define tps1 where tps1 ≡ snd cfg1
  define tps2 where tps2 ≡ snd cfg2
  have cfg1': cfg1' = (st, map (λ(a,tp). act a tp) (zip acts tps1))
    using s1 acts tps1-def sem-def by simp
  have cfg2': cfg2' = (st, map (λ(a,tp). act a tp) (zip acts tps2))
    using s2 cmd2 tps2-def sem-def by simp

  have rs-len: length (config-read cfg1) = ||cfg1||
    by (simp add: read-length)
  have acts-len: length acts = ||cfg1||
    using acts pc rs-len by auto
  hence len-eq: length acts = length tps1
    unfolding tps1-def by simp
  have len-eq2: length acts = length tps2
    using acts-len inv unfolding indist-def tps2-def
    by presburger

  have tapes-gt0: ∀ j>0. snd cfg1' ! j = snd cfg2' ! j
  proof (intro allI impI)
    fix j :: nat assume j-gt: j > 0
    have tps-eq: tps1 ! j = tps2 ! j
      using inv j-gt unfolding indist-def tps1-def tps2-def by blast
    show snd cfg1' ! j = snd cfg2' ! j
    proof (cases j < length acts)
      case True
      have snd cfg1' ! j = act (acts ! j) (tps1 ! j)
          unfolding cfg1' using nth-map-act-zip[OF True len-eq] by (simp add:
tps1-def)
      also have ... = act (acts ! j) (tps2 ! j)
        using tps-eq by simp
```

```
    also have ... = snd cfg2' ! j
      unfolding cfg2' using nth-map-act-zip[OF True len-eq2] by (simp add:
tps2-def)
    finally show ?thesis .
  next
    case False
    then show ?thesis
      using cfg1' cfg2' len-eq len-eq2 nth-overflow by auto
  qed
 qed

 have pre-contents: ∀ j≤C. fst (tps1 ! 0) j = fst (tps2 ! 0) j
  using inv unfolding indist-def tps1-def tps2-def by blast
 have pre-head: snd (tps1 ! 0) = snd (tps2 ! 0)
  using inv unfolding indist-def tps1-def tps2-def
  by (metis hd-span-def)

 have tape0-prefix: ∀ i≤C. (cfg1' <:> 0) i = (cfg2' <:> 0) i
 proof (intro allI impI)
  fix i assume i-le: i ≤ C
  show (cfg1' <:> 0) i = (cfg2' <:> 0) i
  proof (cases acts)
    case Nil
    thus ?thesis unfolding cfg1' cfg2' by simp
  next
    case (Cons a as')
    have idx0: 0 < length acts using Cons by simp
    have (cfg1' <:> 0) i = fst (act (acts ! 0) (tps1 ! 0)) i
      unfolding cfg1' using nth-map-act-zip[OF idx0 len-eq] by (simp add:
tps1-def)
    also have ... = fst (act (acts ! 0) (tps2 ! 0)) i
      using act-prefix-eq[OF pre-contents pre-head] i-le by simp
    also have ... = (cfg2' <:> 0) i
      unfolding cfg2' using nth-map-act-zip[OF idx0 len-eq2] by (simp add:
tps2-def)
    finally show ?thesis .
  qed
 qed

 have hd-eq-le: hd-span cfg1' = hd-span cfg2' ∧ hd-span cfg1' ≤ C
 proof −
  have eq: hd-span cfg1' = hd-span cfg2'
  proof (cases acts)
    case Nil
    thus ?thesis unfolding cfg1' cfg2' hd-span-def by simp
  next
    case (Cons a as')
    have idx0: 0 < length acts using Cons by simp
    have snd cfg1' ! 0 = act (acts ! 0) (tps1 ! 0)
```

```
        unfolding cfg1' using nth-map-act-zip[OF idx0 len-eq] by (simp add:
tps1-def)
      moreover have snd cfg2' ! 0 = act (acts ! 0) (tps2 ! 0)
        unfolding cfg2' using nth-map-act-zip[OF idx0 len-eq2] by (simp add:
tps2-def)
      ultimately show ?thesis
       using act-head-eq[OF pre-head] unfolding hd-span-def by simp
    qed
    moreover have le: hd-span cfg1' ≤ C
      using sem-head-move-bound[OF pc s1] bounds by simp
    ultimately show ?thesis by blast
  qed

  show indist C cfg1' cfg2'
    unfolding indist-def
  proof (intro conjI)
    show ||cfg1'|| = ||cfg2'||
      using len-eq len-eq2
      unfolding cfg1' cfg2'
      by simp
    show fst cfg1' = fst cfg2' using st-eq s1 s2 by simp
    show ∀j>0. snd cfg1' ! j = snd cfg2' ! j using tapes-gt0 .
    show ∀i≤C. (cfg1' <:> 0) i = (cfg2' <:> 0) i using tape0-prefix .
    show hd-span cfg1' = hd-span cfg2' using hd-eq-le by blast
    show hd-span cfg1' ≤ C using hd-eq-le by blast
  qed
qed

lemma indist-execute:
  assumes pm: proper-machine ||cfg|| M
      and inv0: indist C cfg1 cfg2
      and len: ||cfg1|| = ||cfg||
      and hd-bound: hd-span cfg1 + t ≤ C
  shows indist C (execute M cfg1 t) (execute M cfg2 t)
using inv0 hd-bound
proof (induction t)
  case 0
  then show ?case by simp
next
  case (Suc t)
  have t-le: hd-span cfg1 + t ≤ C
    using Suc.prems(2) by simp
  have ind-t: indist C (execute M cfg1 t) (execute M cfg2 t)
    using Suc.IH[OF Suc.prems(1) t-le] .

  define cfg1-t where cfg1-t = execute M cfg1 t
  define cfg2-t where cfg2-t = execute M cfg2 t

  have hd-t-bound: hd-span cfg1-t < C
```

```
proof −
  have pm1: proper-machine ||cfg1|| M
    using pm len by simp
  have hd-span cfg1-t ≤ hd-span cfg1 + t
    unfolding cfg1-t-def
    using execute-head-pos-le-time[OF pm1 refl] by simp
  also have ... < C
    using Suc.prems(2) by simp
  finally show ?thesis .
qed

have exec-Suc-1: execute M cfg1 (Suc t) = exe M cfg1-t
  unfolding cfg1-t-def by simp
have exec-Suc-2: execute M cfg2 (Suc t) = exe M cfg2-t
  unfolding cfg2-t-def by simp

show ?case
proof (cases fst cfg1-t < length M)
  case False
  have step1: exe M cfg1-t = cfg1-t
    unfolding exe-def using False by simp

  have fst cfg2-t = fst cfg1-t
    using ind-t unfolding indist-def
    using cfg1-t-def cfg2-t-def by argo
  hence ¬ (fst cfg2-t < length M)
    using False by simp
  hence step2: exe M cfg2-t = cfg2-t
    unfolding exe-def by simp

  show ?thesis
    unfolding exec-Suc-1 exec-Suc-2
    using cfg1-t-def cfg2-t-def ind-t step1 step2
    by auto
next
  case True
  define cmd where cmd = M ! fst cfg1-t

  have step1: exe M cfg1-t = sem cmd cfg1-t
    unfolding exe-def cmd-def using True by simp

  have fst cfg2-t = fst cfg1-t
    using ind-t unfolding indist-def
    using cfg1-t-def cfg2-t-def by presburger
  hence step2: exe M cfg2-t = sem cmd cfg2-t
    unfolding exe-def cmd-def using True by simp

  have tapes-eq: ||cfg1-t|| = ||cfg||
    unfolding cfg1-t-def using execute-num-tapes-proper len
```

```
    using pm by presburger
  have pm-t: proper-machine ||cfg1-t|| M
    using pm tapes-eq by simp
  have pc: proper-command ||cfg1-t|| cmd
    using pm-t True cmd-def by simp

  have indist C (sem cmd cfg1-t) (sem cmd cfg2-t)
    using cfg1-t-def cfg2-t-def hd-t-bound ind-t indist-step pc
    by blast
  thus ?thesis
    unfolding exec-Suc-1 exec-Suc-2
    using cfg1-t-def cfg2-t-def hd-t-bound ind-t indist-step pc
    using step1 step2 by argo
 qed
qed
```

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 :: nat  and w v :: string  and G :: nat and M :: machine
 defines cfgw ≡ start-config-string k w
    and cfgv ≡ start-config-string k v
 assumes TM  : turing-machine k G M
    and EQ   : take (Suc C) w = take (Suc C) v
    and LE   : t ≤ C
 shows (execute M cfgw t <:> 1) = (execute M cfgv t <:> 1)
proof −
 have k-ge-2: 2 ≤ k
   using TM unfolding turing-machine-def by simp

 have len-w: ||cfgw|| = k
   unfolding cfgw-def using start-config-length k-ge-2
   by (metis bot-nat-0.not-eq-extremum
       not-numeral-le-zero)
 have len-v: ||cfgv|| = k
   unfolding cfgv-def using start-config-length k-ge-2
   by (metis bot-nat-0.not-eq-extremum
       not-numeral-le-zero)
 have len-eq: ||cfgw|| = ||cfgv||
   using len-w len-v by simp

 have pm: proper-machine ||cfgw|| M
   using TM len-w unfolding turing-machine-def
   using nth-mem turing-commandD(1) by blast

 have inv0: indist C cfgw cfgv
```

```
    unfolding cfgw-def cfgv-def
    using indist-start[OF EQ k-ge-2] .

  have hd-0: hd-span cfgw = 0
   unfolding cfgw-def start-config-string-conv hd-span-def start-config-def by simp
  have t-bound: hd-span cfgw + t ≤ C
    using hd-0 LE by simp

  have ind-t: indist C (execute M cfgw t) (execute M cfgv t)
    using indist-execute inv0 pm t-bound by blast

  have tape1-idx: 1 > (0::nat) by simp
  have snd (execute M cfgw t) ! 1 = snd (execute M cfgv t) ! 1
    using ind-t tape1-idx unfolding indist-def by presburger

  thus ?thesis
    by presburger
qed

lemma string-to-contents-inj:
  assumes string-to-contents xs = string-to-contents ys
  shows   xs = ys
proof −
  have eq-contents: ⌊string-to-symbols xs⌋ = ⌊string-to-symbols ys⌋
    using assms contents-string-to-contents by simp
  have len: length xs = length ys
  proof (rule ccontr)
    assume length xs ≠ length ys
    then consider (less-xs) length xs < length ys | (less-ys) length ys < length xs
      by linarith
    then show False
    proof cases
      case less-xs
      define i where i = Suc (length xs)

      have ⌊string-to-symbols xs⌋ i = 0
        unfolding i-def by simp
      moreover have ⌊string-to-symbols ys⌋ i ≠ 0
        using less-xs i-def by simp

      ultimately show False
        using eq-contents by simp
    next
      case less-ys
      define i where i = Suc (length ys)

      have ⌊string-to-symbols ys⌋ i = 0
        unfolding i-def by simp
      moreover have ⌊string-to-symbols xs⌋ i ≠ 0
```

```
      using less-ys i-def by simp

    ultimately show False
      using eq-contents by simp
  qed
 qed
 moreover have xs ! j = ys ! j if j < length xs for j
 proof −
  have ⌊string-to-symbols xs⌋ (Suc j) = ⌊string-to-symbols ys⌋ (Suc j)
    using eq-contents by auto
  thus ?thesis
    using that len
    using Suc-1 Suc-diff-1 Suc-le-eq by force
 qed
 ultimately show ?thesis
  by (rule nth-equalityI)
qed

lemma constant-time-depends-on-prefix:
 fixes f :: string ⇒ string
 assumes TM: turing-machine k G M
    and CT: computes-in-time k M f (λ-. C)
    and EQ: take (Suc C) w = take (Suc C) v
 shows f w = f v
proof −
 define cfgw where cfgw = start-config-string k w
 define cfgv where cfgv = start-config-string k v

 have out-w: execute M cfgw C <:> 1 = string-to-contents (f w)
  using computes-in-time-execute[OF CT, of w] unfolding cfgw-def
  by blast

 have out-v: execute M cfgv C <:> 1 = string-to-contents (f v)
  using computes-in-time-execute[OF CT, of v] unfolding cfgv-def
  by blast

 have tapes-eq: execute M cfgw C <:> 1 = execute M cfgv C <:> 1
   using exec-same-output-C[where k=k and C=C and t=C and w=w and
v=v and G=G and M=M]
  using TM EQ le-refl unfolding cfgw-def cfgv-def
  by blast

 have string-to-contents (f w) = string-to-contents (f v)
  using out-w out-v tapes-eq
  by presburger

 thus f w = f v
  by (rule string-to-contents-inj)
qed
```

```
lemma DTIME1-depends-on-prefix:
  fixes L :: language
  assumes L ∈ DTIME (λn. 1)
  obtains C H where ∀ w. w ∈ L ⟷ take (Suc C) w ∈ H
proof −
  obtain k G M C where tm: turing-machine k G M
    and comp: computes-in-time k M (characteristic L) (λ-. C)
    using DTIME1-const-time[OF assms] by blast

  let ?H = {u. ∃ w. u = take (Suc C) w ∧ w ∈ L}

  have ∀ w. w ∈ L ⟷ take (Suc C) w ∈ ?H
  proof
    fix w show w ∈ L ⟷ take (Suc C) w ∈ ?H
    proof
      assume w ∈ L
      then show take (Suc C) w ∈ ?H by auto
    next
      assume take (Suc C) w ∈ ?H
      then obtain v where eqpref: take (Suc C) w = take (Suc C) v and v ∈ L
        by auto

      have characteristic L w = characteristic L v
        using constant-time-depends-on-prefix[OF tm comp eqpref] .

      thus w ∈ L
        using ‹v ∈ L› unfolding characteristic-def by simp
    qed
  qed

  thus ?thesis
    by (rule that[of C ?H])
qed
```

1.4 Main Theorem: SAT $\notin$ 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 ¬ (∃ C H. ∀ w. w ∈ SAT ⟷ take (Suc C) w ∈ H)
proof
  assume ∃ C H. ∀ w. w ∈ SAT ⟷ take (Suc C) w ∈ H
  then obtain C H where prefix-decider: ∀ w. w ∈ SAT ⟷ take (Suc C) w ∈ H
```

by *blast*

define φ-*unsat* :: *formula* **where** φ-*unsat* = *replicate* (*Suc C*) []
have ¬ *satisfiable* φ-*unsat*
unfolding φ-*unsat-def satisfiable-def satisfies-def satisfies-clause-def*
by *simp*

define *v* **where** *v* = *formula-to-string* φ-*unsat*

have *v* ∉ *SAT*
unfolding *SAT-def v-def* **using** ‹¬ *satisfiable* φ-*unsat*› *formula-to-string-inj* **by** *auto*

have *len-v*: *length v* ≥ *Suc C*
proof −
have *formula-n* φ-*unsat* = *replicate* (*Suc C*) []
unfolding φ-*unsat-def formula-n-def clause-n-def* **by** *simp*
hence *nlllength* (*formula-n* φ-*unsat*) = *nlllength* (*replicate* (*Suc C*) [])
by *simp*

also have ... = *nllength* (*concat* (*replicate* (*Suc C*) ([]::*nat list*))) + *length* (*replicate* (*Suc C*) ([]::*nat list*))
using *nlllength-nllength-concat*
by *simp*
also have ... = *nllength* ([]::*nat list*) + *Suc C*
by *simp*
also have ... = *Suc C*
by *simp*
finally have *inner-len*: *nlllength* (*formula-n* φ-*unsat*) = *Suc C* **.**

have *length v* = *length* (*symbols-to-string* (*binencode* (*numlistlist* (*formula-n* φ-*unsat*))))
unfolding *v-def* **by** *simp*
also have ... = *length* (*binencode* (*numlistlist* (*formula-n* φ-*unsat*)))
by *simp*
also have ... = *2* ∗ *nlllength* (*formula-n* φ-*unsat*)
unfolding *nlllength-def* **by** *simp*
also have ... = *2* ∗ *Suc C*
using *inner-len* **by** *simp*

finally show *?thesis*
by *simp*
qed

define *w* **where** *w* = (*if odd* (*Suc C*) *then take* (*Suc C*) *v else take* (*Suc C*) *v* @ [*v* ! *0*])

have *pref-eq*: *take* (*Suc C*) *w* = *take* (*Suc C*) *v*
unfolding *w-def* **using** *len-v* **by** *auto*

```
  have odd-w: odd (length w)
    using len-v unfolding w-def by auto

  have w ∈ {x. ¬ (∃φ. x = formula-to-string φ)}
  proof (rule ccontr)
    assume ¬ w ∈ {x. ¬ (∃φ. x = formula-to-string φ)}
    then obtain φ' where w-phi: w = formula-to-string φ' by auto

    have even (length w)
      unfolding w-phi
      by simp

    with odd-w show False by simp
  qed

  then have w ∈ SAT
    unfolding SAT-def by blast

  have take (Suc C) w ∈ H ⟷ take (Suc C) v ∈ H
    using pref-eq by simp

  moreover have w ∈ SAT ⟷ take (Suc C) w ∈ H
    using prefix-decider by simp

  moreover have v ∈ SAT ⟷ take (Suc C) v ∈ H
    using prefix-decider by simp

  ultimately show False
    using ‹w ∈ SAT› ‹v ∉ SAT› by simp
qed

theorem SAT-not-const-time:
  shows SAT ∉ DTIME (λn. 1)
proof
  assume SAT ∈ DTIME (λn. 1)

  then obtain C H where ∀ w. w ∈ SAT ⟷ take (Suc C) w ∈ H
    using DTIME1-depends-on-prefix by blast

  with SAT-not-prefix-language show False
    by blast
qed

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.