A Deeper Look at Depth: Stable Generation Accounting for Quantifier Reasoning

Can Cebeci[1], Nikolaj Bjørner[2], George Candea[1], and Clément Pit-Claudel[1]

[1] EPFL, Lausanne, Switzerland
[2] Microsoft Research, Redmond, USA

Abstract. SMT solvers make automated verification convenient. At the same time, solvers suffer from instability, whereby seemingly inconsequential changes to the input may cause a previously quickly produced proof to fail or time out. This paper addresses a common cause of outcome instability (i.e., unsat/unknown fluctuations) in the context of program verification. We demonstrate that the generation (i.e., depth) accounting used to make quantifier instantiation practical and implemented in multiple state-of-the-art SMT solvers is non-confluent (i.e., prone to divergence), and that this leads to instability. We then address this deficiency and develop a new accounting method that is stable. The new method is justified using a sequence of refinements from an abstract, confluent solver model all the way to our implementation in Z3. Our empirical evaluation demonstrates that our implementation reduces outcome instability by 94% in the unstable core of the Mariposa benchmark without leading to performance regressions.

Keywords: SMT solving · instability · quantifier reasoning

1 Introduction

SMT solvers make automated verification convenient but brittle. They automatically discharge most proof obligations, significantly lowering the required effort and expertise for developing proofs. At the same time, solvers suffer from instability, whereby seemingly inconsequential changes to the input may cause a previously quickly produced proof to fail or time out [16]. Recent work [4] conjectures that the instability users encounter in practice is addressable, not fundamental. This paper addresses a common cause of outcome instability (i.e., `unsat` queries turning `unknown`) in the context of program verification.

Controlling quantifier instantiation is crucial for stability [15, 17, 9]. This is because quantifiers are heavily used to encode verification conditions, and the solver's success depends on choosing the right instances: under-instantiation leads to `unknown` and over-instantiation leads to `timeout`.

Program verifiers typically control instantiation by configuring the solver to instantiate based on E-matching [10] exclusively, and up to some form of depth threshold [8, 7, 14]. For Z3 [11], they set `smt.qi_eager_threshold`. For CVC5 [1],

`inst-max-level`. Following Z3's terminology, we refer to measures of a term's instantiation depth as its generation number.

We demonstrate that the generation accounting implemented in some of the state-of-the-art SMT solvers is non-confluent (i.e., prone to divergence), and that this leads to instability. More precisely, we show that in certain cases a term's generation number (and subsequently the solver's success) is dependent on the order in which the solver explores proof branches. These cases are not simple solver bugs. Rather, the ambiguity is conceptual: to the best of our knowledge, there is no precise definition of what generation numbers should represent. We use the terminology confluence here in the sense of Church-Rosser [5] properties. It is fundamental in automated deduction, where a terminating and confluent set of rewrite rules ensures decidability [6, 3] in several settings. SMT solvers using E-matching for quantifier instantiation ensure termination by bounding quantifier generation. But as we observe, bounding instantiation using generation thresholds is an important ingredient for breaking confluence.

We address this issue with the following contributions:

- A precise, solver-agnostic specification of generation semantics for E-matching with quantifier weights.
- Id: an abstract state machine model of solver behavior with access to idealized generation numbers. Id is confluent, but not practically implementable. It serves as a reference for the generation accounting problem independently of a specific implementation.
- N, T, C: a series of implementable, increasingly efficient solver models, each obtained by refining the previous one. We prove, through bisimulation, that each solver is confluent and no more permissive than Id with quantifier instantiation.
- An efficient implementation of C in Z3, upstream as of version 5.0.
- An empirical evaluation demonstrating that the implementation reduces outcome instability by 94%, while incurring 2% fewer timeouts, in the unstable core of the Mariposa benchmark [16], which includes program verification queries. We also demonstrate that our implementation does not lead to performance regressions for SMT-LIB benchmarks [13].

In this work-in-progress paper, we present proof sketches for the refinements N, T, and C, aiming to convey the underlying intuition rather than full rigor. We present some of the fully formalized proofs in the appendix; the correctness of the remaining refinements is conjectured, supported by the proof sketches.

2 Background and Motivating Examples

In this section, we provide an overview of E-matching with generation thresholds and quantifier weights, and motivate the need to precisely define generation semantics.

We use the following query as a running example:

```
forall x. f(x+1) > f(x)
f(0) > 0
f(3) < 0
```

The query is unsatisfiable: instantiating the quantifier three times with $x := 0, 1, 2$ leads to a contradiction.

2.1 E-matching

Program verifiers typically configure SMT solvers to produce quantifier instantiations using E-matching, whereby the solver produces instances based on ground terms that match a given pattern. For our running example, the pattern $f(x)$ would lead to the right instantiations: $f(0)$ in the input would match $f(x)$ with $x := 0$. That instantiation would create the term $f(1)$, which would match again with $x := 1$ and so on.

Though the pattern is productive in this scenario, it also creates a matching loop. Had the query been satisfiable (e.g., had the last assertion been omitted), the solver could produce infinitely many instances rather than quickly giving up. To guard against this, most solvers expose configuration parameters that set thresholds controlling when to give up and return `unknown`.

2.2 Generation thresholds

A common idea is to threshold the *depth* of instantiation chains as opposed to the total number of instantiations. The latter is prone to brittleness, as a productive instantiation may never take place depending on the order in which the solver explores proof branches.

One way to track instantiation depth is to associate a generation number with each term, such that all input terms have generation 0, and terms created through an instantiation have 1 plus the maximal generation among matching terms [2]. We block an instantiation if the maximal generation is above the threshold. In the example above $f(0)$ would have generation 0, $f(1)$ would have 1 and so on. Therefore, a generation threshold of 3 would suffice for the solver to return `unsat`.

2.3 Quantifier weights

Now consider adding another quantifier to the query:

```
forall x. f(f(x)) > f(x+2) > f(x) ::: pattern f(x)
```

This quantifier is more expensive: it creates two fresh terms, and both terms re-match the pattern. So it produces 2^{c-1} instantiations where c is the generation

threshold. We would like to keep instantiation chains involving this quantifier shorter than others. One way to do this is to associate a larger *weight* with expensive quantifiers, and increment the generation numbers of fresh terms by the weight, instead of just by 1.

To prevent recursive matches on the expensive quantifier while still allowing the productive instantiations, we can configure the query as follows:

```
generation threshold: 3
forall x. f(f(x)) > f(x+2) > f(x) ::: pattern: f(x), weight: 3
forall x. f(x+1) > f(x) ::: pattern: f(x), weight: 1
f(0) > 0
f(3) < 0
```

2.4 Generation ambiguity

Example 1 In the current state of our working example, the generation number of $f(2)$ is ambiguous: $f(0)$ is part of the input so it has generation 0. Then the first quantifier can generate $f(2)$ at generation 3. On the other hand the second quantifier can generate $f(2)$ at generation 2 after two rounds of instantiation. This ambiguity gives rise to instability, since $f(2)$'s generation determines whether the instantiation producing $f(3) > f(2)$ is allowed, and consequently whether the query is `unsat` or `unknown`.

Example 2 The ambiguity is not uniquely brought on by quantifier weights. Consider the following query:

```
generation threshold: 3
forall x. g(x) == h(x) ::: pattern g(x), weight 1
forall x. h(x) == g(x+1) ::: pattern h(x), weight 1
forall x. f(g(x)) > g(x) ::: pattern g(x), weight 1
forall x. f(f(x)) > f(x) ::: pattern f(x), weight 1
forall x. f(f(x)) < 0 ::: pattern f(f(x)), weight 1
g(0) > 0
```

$g(0)$ has generation 0. By instantiating the first two quantifiers, we can generate $g(1)$ at generation 2, and learn $g(0) = g(1)$. From the third quantifier, we get $f(g(0))$ at generation 1.

The fourth quantifier should either match $f(g(0))$ or $f(g(1))$, the latter through the equality we learned. Matching both would be wasteful, since $x := g(0)$ and $x := g(1)$ are equivalent bindings. If the fourth quantifier matches $f(g(0))$, we get $f(f(g(0)))$ at generation 2, which matches the fifth quantifier and leads to a contradiction, returning `unsat`. But if the fourth quantifier matches $f(g(1))$, since $g(1)$ has generation 2 , we get $f(f(g(1)))$, an equivalent term, at generation 3 and so give up, returning `unknown`, without instantiating the fifth quantifier.

3 Preliminaries

3.1 Terms and contexts

Let x, c, and f range over variables, constants, and uninterpreted function symbols, respectively. Terms are given by the grammar

$$t := x \mid c \mid f(t_1, \ldots, t_n).$$

A context is a term containing exactly one hole $\Box$, which is a proper subterm. Contexts are defined recursively by

$$C := f(t_1, \ldots, \Box, \ldots, t_n) \mid f(t_1, \ldots, C, \ldots, t_n).$$

For a context C and term t, we write $C[t]$ for the term obtained by replacing the unique occurrence of $\Box$ in C with t.

3.2 Congruence and equality

We make a central distinction between equivalence and congruence modulo the theory of equality. Equivalence is a weaker notion: two expressions s, t, are equivalent in E, written $t \cong_E s$, if the equality between s and t can be derived from equalities in E using any rule in the theory of equality (reflexivity, symmetry, transitivity, congruence). On the other hand, two terms are congruent if they are equivalent and their equivalence can be inferred using the congruence rule:

Definition 1 (Congruence between terms). *We say that t and t' are congruent if they have the same function symbol and all arguments are equivalent in E. In other words, for every function symbol f, congruence is given by*

$$\frac{t_i \cong_E t'_i \text{ for } i = 1, \ldots, n}{\mathit{congruent}_E(f(t_1, \ldots, t_n), f(t'_1, \ldots, t'_n))}$$

The E-matching and congruence-closure literature [12] uses terminology that can blur this distinction. For example, the statement "if two terms $f(t_1, \ldots, t_n)$ and $f(t'_1, \ldots, t'_n)$ are congruent, then it is wasteful to try to match both" [10] is not valid under the weaker interpretation where only $f(a) = f(b)$ is known: matching $f(x)$ can produce non-equivalent bindings $x := a$, $x := b$ when $a \neq b$.

3.3 Quantifiers and instances

Quantifiers and quantifier instances are not terms. Instead, we treat them as distinct syntactic objects.

For simplicity, we assume each quantifier binds a single pattern and that quantifiers are not nested. Generalizing to multiple bound variables and nested quantifiers is conceptually straightforward but at the cost of more involved formalization.

A quantifier instance is denoted by $q[\![\theta]\!]$, where q is a quantifier and θ is the term matching its pattern (not the bound variable). We write $t \in q[\![\theta]\!]$ to mean that the instantiated body of q with θ contains t as a subterm.

Each quantifier q is associated with a non-negative integer weight, denoted by $w(q)$.

3.4 Other notation

- For functions f, g, h and predicate P, $f' = f[g(x) \mapsto h(x) \mid P(g(x))]$ denotes $\forall x.\ (P(g(x)) \implies f'(g(x)) = h(x)) \wedge \forall y.\ (\neg P(y) \implies f'(y) = f(y))$
- Similarly $f' = f[y \mapsto z]$ denotes $f'(y) = z \wedge \forall x.\ x \neq y \implies f'(x) = f(x)$

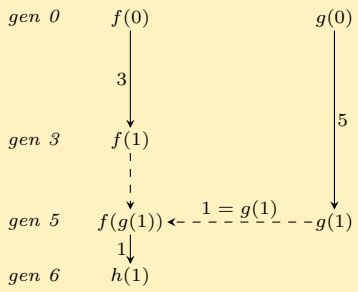
4 Defining Generation Semantics

Intuitively, we define the generation number of a term as the weighted depth of the shallowest derivation tree, where leaves are input terms and edges are quantifier instantiations or equality rewrites using previously-derived terms.

For intuition, consider the following scenario. Solid lines represent instantiations and dashed lines represent rewrites:

- input terms: $f(0)$ and $g(0)$
- quantifiers ($pattern \xrightarrow{weight} term$): $f(x) \xrightarrow{3} f(x+1)$, $g(x) \xrightarrow{5} g(x+1)$, $f(g(x)) \xrightarrow{1} h(x)$
- equality $g(1) = 1$

The generation number of $h(1)$ is 6, through the derivation tree shown.



Definition 2 (Reachable generation). *Given a set of quantifiers Q, input terms A, and equalities E, and instantiations I, a reachable generation g of a term t is a tuple $t \vdash_{E,I} g$, that can be obtained from one of the inferences:*

$$\frac{t \in A}{t \vdash_{E,I} 0}\ \textit{(Input)}$$

$$\frac{q \in Q \qquad t \in q[\![\theta]\!] \in I \qquad \theta \vdash_{E,I} g}{t \vdash_{E,I} w(q) + g}\ \textit{(Inst)}$$

$$\frac{s_1 \cong_E s_2 \qquad t[s_1] \vdash_{E,I} g \qquad s_2 \vdash_{E,I} g'}{t[s_2] \vdash_{E,I} \max(g, g')}\ \textit{(Congruence)}$$

A and Q are constant given an input query, so we assume them implicit arguments. A is subterm-closed (i.e., $\forall t.\ t \in A \wedge t' \in t \implies t' \in A$)

It is not practically feasible for solver implementations to track generation numbers in a way that exhaustively considers all possible quantifier instantiations. We therefore make a distinction between ideal and implementation generations.

Definition 3 (Implementation generation). *We define the implementation generation of a term t with respect to A, Q, E, I as*

$$\mathrm{gen}_{E,I}(t) := \min\{g \mid t \vdash_{E,I} g\}$$

Definition 4 (Universal set of instantiations).

$$I^{\infty} := \{q[\![\theta]\!] \mid q \in Q, \theta \textit{ is a term}\}$$

Definition 5 (Ideal generation). *We define the ideal generation of a term t with respect to A, Q, E as*

$$\mathrm{gen}_E^{\mathsf{ideal}}(t) := \min\{g \mid t \vdash_{E,I^{\infty}} g\}$$

Whenever convenient, we denote $\mathrm{gen}_E^{\mathsf{ideal}}(t)$ with $\mathrm{gen}_{E,I^{\infty}}(t)$

For a set of equalities E, $\mathrm{gen}_E^{\mathsf{ideal}}(t)$ is uniquely determined and independent of instantiation history, backtracking, or internal data-structure state.

5 An Ideal Solver Model - Id

This section presents Id, a solver model with access to $\mathrm{gen}_E^{\mathsf{ideal}}(t)$, used as a reference for practical algorithms.

Our method only depends on inference steps involving quantifier instantiations and equalities, and the model omits preconditions and inferences that are irrelevant to us. In particular, we do not model conflicts, instead assuming the solver may backtrack at any point.

5.1 State representation

The ideal solver maintains a state of the form $\Sigma_{\mathsf{Id}} : \langle \tau \rangle$ where

- τ : Event* - stack of locally asserted equalities and instances
- Event := $\mathtt{eq}(t_1 \simeq t_2) \mid \mathtt{inst}(q[\![\theta]\!])$ for $q \in Q$

The initial state is, $\sigma_{\mathsf{Id}}^{init} := \langle \tau := \langle\rangle \rangle$.

Beyond the current state $\langle \tau \rangle$ we assume the following global constants:

- A : Set(Term) (ground terms in input query),
- Q : Set(Quantifier) (quantifiers),
- $\mathsf{maxG} : \mathbb{N}$ (generation threshold).

For convenience we also define a few shortcuts based on a state:

- $E(\tau) := \{e \mid \texttt{eq}(e) \in \tau\}$: Set(Term) - set of equalities that are asserted within the scope.
- $I(\tau) := \{q[\![\theta]\!] \mid \texttt{inst}(q[\![\theta]\!]) \in \tau\}$: Set(Instance) - set of active quantifier instantiations.

Unless indicated otherwise, we use E as shorthand for $E(\tau)$ and I for $I(\tau)$. We also use $\mathcal{T}(\sigma)$ to refer to τ where $\sigma := \langle\tau\rangle$.

Definition 6 (Enabled instances). *For a set of equalities E and quantifier instances I, define the set of instances enabled within generation threshold* maxG *as*

$$\varepsilon_E(I) := \{q[\![\theta]\!] \mid w(q) + \text{gen}_{E,I}(\theta) \leq \mathsf{maxG},\ q \in Q,\ \theta \text{ is a term}\}$$

The transition rules for Id are as follows:

$$\frac{e := (t_1 \simeq t_2)}{\langle\tau\rangle \rightarrow_{\mathsf{Id}} \langle\tau \cdot \texttt{eq}(e)\rangle}\ (\text{PushEq}) \qquad \frac{}{\langle\tau \cdot \texttt{eq}(e)\rangle \rightarrow_{\mathsf{Id}} \langle\tau\rangle}\ (\text{PopEq})$$

$$\frac{q[\![\theta]\!] \in \varepsilon_E(I^\infty)}{\langle\tau\rangle \rightarrow_{\mathsf{Id}} \langle\tau \cdot \texttt{inst}(q[\![\theta]\!])\rangle}\ (\text{PushInst}) \qquad \frac{}{\langle\tau \cdot \texttt{inst}(q[\![\theta]\!])\rangle \rightarrow_{\mathsf{Id}} \langle\tau\rangle}\ (\text{PopInst})$$

We use $\rightsquigarrow_{\mathsf{Id}}$ to denote the transitive closure of $\rightarrow_{\mathsf{Id}}$ (i.e., reachability).

5.2 Stability and confluence

Our goal is to prove that each solver is stable, i.e., if there is some way for the solver to return `unsat`, the solver will always do so. We do not model decisions and conflicts, or returning `unsat` or `unknown`. Instead we provide a definition of confluence and argue that a confluent solver is outcome-stable.

A solver returns `unsat` if it finds a set of conflicts that suffice to prove false, and returns `unknown` if no such set can be found. When the solver visits $\langle\tau\rangle$, it may find a conflict between equalities in $E(\tau)$ considering instances in $I(\tau)$. We define all such (potentially conflicting) contexts the solver may observe starting at state σ as $\text{Obs}(\sigma)$:

Definition 7 (Observable contexts).

$$\text{Obs}(\sigma) := \{\langle E_o, I_o\rangle \mid \sigma \rightsquigarrow_S \sigma' \wedge E_o \subseteq E(\mathcal{T}(\sigma')) \wedge I_o \subseteq I(\mathcal{T}(\sigma'))\}$$

We call a solver confluent if choosing to take certain steps never shrinks the set of observable contexts.

Definition 8 (Confluence). *Solver S with initial state σ_S^{init} is confluent if for all states σ*

$$\sigma_S^{init} \rightsquigarrow_S \sigma \implies \text{Obs}(\sigma_S^{init}) = \text{Obs}(\sigma)$$

A confluent solver is outcome-stable: if the solver may return `unsat`, there is a set of conflicts, each observable from the initial state, that collectively prove `unsat`. Due to confluence those conflicts remain observable after every transition, so the solver will never return `unknown`.

Theorem 1. Id *is confluent.*

Proof. Id can perfectly backtrack every step, so it is always possible to return from σ to σ_S^{init}, and from there observe the same contexts.

6 Realizable (And Efficient) Solver Models

Here we present our algorithm through a sequence of realizable solver models that get iteratively more efficient. §7 then describes how the last algorithm translates into a Z3 implementation.

For each solver S, we prove that S is confluent and that S is not more permissive with quantifier instantiation than Id. We do so by defining a simulation relation between solvers, proving bisimulation between Id and each subsequent solver, and proving lemmas which state that bisimulation implies the desired properties (Lemma 1, Lemma 2).

The state tuple of each solver model includes τ, the event trail. τ is always initially empty, and PushEq is always available. We define a notion of subsumption between trails, which we then use to define simulation between solvers:

Definition 9 (Trail subsumption). *We define subsumption between trails τ, τ' as*

$$\tau \sqsubseteq \tau' := E(\tau) \subseteq E(\tau') \wedge I(\tau) \subseteq I(\tau')$$

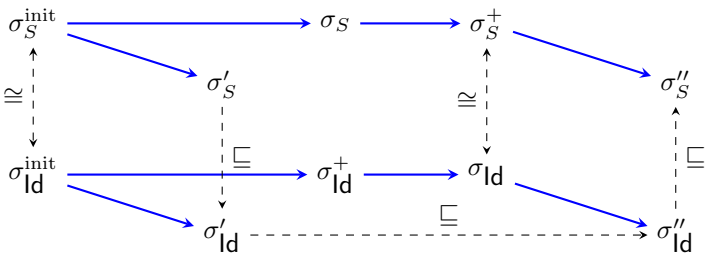
Definition 10 (Simulation). *We define simulation between solvers R,S as*

$$R \preceq S := \forall \sigma_S, \sigma_R, \sigma'_R.\ \sigma_S^{init} \rightsquigarrow_S \sigma_S \wedge \sigma_R^{init} \rightsquigarrow_R \sigma_R \wedge \mathcal{T}(\sigma_R) \sqsubseteq \mathcal{T}(\sigma_S) \wedge \sigma_R \rightarrow_R \sigma'_R$$
$$\implies \exists \sigma'_S.\ \sigma_S \rightsquigarrow_S \sigma'_S \wedge \mathcal{T}(\sigma'_R) \sqsubseteq \mathcal{T}(\sigma'_S)$$

Lemma 1 (Simulation preserves observations). *For solvers R and S, if $S \preceq R$,* $\mathrm{Obs}(\sigma_S^{init}) \subseteq \mathrm{Obs}(\sigma_R^{init})$.

Proof. Given $\langle E_o, I_o \rangle \in \mathrm{Obs}(\sigma_S^{init})$, we know there is a σ_S with $\sigma_S^{init} \rightsquigarrow_S \sigma_S$ and $E_o \subseteq E(\sigma_S) \wedge I_o \subseteq I(\sigma_S)$. We have $\mathcal{T}(\sigma_S^{init}) \sqsubseteq \mathcal{T}(\sigma_R^{init})$ since both trails are empty. Since $S \preceq R$, we can simulate every step in $\sigma_S^{init} \rightsquigarrow_S \sigma_S$ to get a σ_R with $\sigma_R^{init} \rightsquigarrow_R \sigma_R$ and $\mathcal{T}(\sigma_S) \sqsubseteq \mathcal{T}(\sigma_R)$. Then we have $E_o \subseteq E(\sigma_R) \wedge I_o \subseteq I(\sigma_R)$ and so $\langle E_o, I_o \rangle \in \mathrm{Obs}(\sigma_R^{init})$.

Lemma 2 (Bisimulation with Id implies confluence). *For solver S, if* $\mathsf{Id} \preceq S$ *and* $S \preceq \mathsf{Id}$, *S is confluent.*



Proof. Given $\sigma_S^{init} \rightsquigarrow_S \sigma_S$, and a $\langle E_o, I_o\rangle \in \text{Obs}(\sigma_S^{init})$, it suffices to show $\langle E_o, I_o\rangle \in \text{Obs}(\sigma_S)$, since we trivially have $\text{Obs}(\sigma_S) \subseteq \text{Obs}(\sigma^{init})$.

$\mathcal{T}(\sigma_S^{init}) \sqsubseteq \mathcal{T}(\sigma_{\mathsf{Id}}^{init})$ since both stacks are empty. Then through $S \preceq \mathsf{Id}$, we have a σ_{Id}^+ with $\sigma_{\mathsf{Id}}^{init} \rightsquigarrow_{\mathsf{Id}} \sigma_{\mathsf{Id}}^+$ and $\mathcal{T}(\sigma_S) \sqsubseteq \mathcal{T}(\sigma_{\mathsf{Id}}^+)$. Since Id can fully backtrack and push the same equalities with fewer instantiations, we also have a σ_{Id} with $\sigma_{\mathsf{Id}}^{init} \rightsquigarrow_{\mathsf{Id}} \sigma_{\mathsf{Id}}^+ \rightsquigarrow_{\mathsf{Id}} \sigma_{\mathsf{Id}}$, $I(\sigma_S) = I(\sigma_{\mathsf{Id}})$, and $E(\sigma_S) \subseteq E(\sigma_{\mathsf{Id}})$. And since pushing equalities is always allowed, σ_S can transition to a σ_s^+ that includes the missing equalities, and so $\mathcal{T}(\sigma_{\mathsf{Id}}) \sqsubseteq \mathcal{T}(\sigma_s^+)$

By the definition of Obs(), there is a σ'_S with $\sigma_S^{init} \rightsquigarrow_S \sigma'_S$, $E_o \subseteq E(\mathcal{T}(\sigma'_S))$ and $I_o \subseteq I(\mathcal{T}(\sigma'_S))$. Since $S \preceq \mathsf{Id}$, we have a σ'_{Id} with $\sigma_{\mathsf{Id}}^{init} \rightsquigarrow_{\mathsf{Id}} \sigma'_{\mathsf{Id}}$ and $\mathcal{T}(\sigma'_S) \sqsubseteq \mathcal{T}(\sigma'_{\mathsf{Id}})$.

Since Id is confluent, we have a σ''_{Id} with $\sigma_{\mathsf{Id}} \rightsquigarrow_{\mathsf{Id}} \sigma''_{\mathsf{Id}}$ and $E(\mathcal{T}(\sigma''_{\mathsf{Id}})) \supseteq E(\mathcal{T}(\sigma'_{\mathsf{Id}})) \supseteq E_o$ and similarly $I(\mathcal{T}(\sigma''_{\mathsf{Id}})) \supseteq I_o$. Then, since $\mathsf{Id} \preceq S$ and $\mathcal{T}(\sigma_{\mathsf{Id}}) \sqsubseteq \mathcal{T}(\sigma_S)$, there is a σ''_S with $\sigma_S \rightsquigarrow_S \sigma''_S$ and $\mathcal{T}(\sigma''_{\mathsf{Id}}) \sqsubseteq \mathcal{T}(\sigma''_s)$. Then $E_o \subseteq E(\mathcal{T}(\sigma''_{\mathsf{Id}}))$ and $I_o \subseteq I(\mathcal{T}(\sigma''_{\mathsf{Id}}))$ and so $\langle E_o, I_o\rangle \in \text{Obs}(\sigma_S)$.

6.1 Refinement 1: A Naive Solver – N

The most straightforward (and slow) approach is to compute all generation numbers at matching time using $I(\tau)$, the current instance set.

The naive solver maintains the same state as the ideal. It uses $\varepsilon_E(I)$ instead of $\varepsilon_E(I^\infty)$:

$$\frac{q[\![\theta]\!] \in \varepsilon_E(I)}{\langle\tau\rangle \rightarrow_{\mathsf{N}} \langle\tau \cdot \texttt{inst}(q[\![\theta]\!])\rangle} \text{ (PushInst}_{\mathsf{N}}\text{)}$$

All other transitions remain unchanged.

Theorem 2. $\mathsf{Id} \preceq \mathsf{N}$.

Proof. PushEq can be trivially simulated by the corresponding step in N. PopEq and PopInst can be simulated without any transitions. PushInst can be simulated by a sequence of $\text{PushInst}_{\mathsf{N}}$ steps: one for each quantifier instance used in the inference of the matching term's ideal generation, starting at the leaves. We present a detailed proof for the PushInst case in Appendix A.

Theorem 3. $\mathsf{N} \preceq \mathsf{Id}$

Proof. $\text{PushEq}_{\mathsf{N}}$ and $\text{PushInst}_{\mathsf{N}}$ can each be simulated by the corresponding step in Id. $\text{PopEq}_{\mathsf{N}}$ and $\text{PopInst}_{\mathsf{N}}$ can be simulated without any transitions.

6.2 Refinement 2: Generation-Tracking Solver – T

The next step is to avoid recomputing $\text{gen}_{E,I}(t)$ from scratch at matching time. Instead, we maintain a dynamic mapping from terms to generation numbers.

Generation tracking states are of the form $\Sigma_{\mathsf{T}} : \langle\tau, g\rangle$ where

- τ, E, I have the same meaning as in Σ_{Id}
- $g : \text{Term} \to \mathbb{N}$ - mapping used to track generation numbers

Initially, $g(t) = 0$ if $t \in A$, and $g(t) = \infty$ otherwise.

Handling equivalent instances: Real-world solvers avoid creating equivalent instantiations: given θ_1, θ_2 with $\mathit{congruent}_E(\theta_1, \theta_2)$, for any q, the solver instantiates at most one of $q[\![\theta_1]\!]$ or $q[\![\theta_2]\!]$, since it judges the two instances to be equivalent. This complicates generation tracking, as the generation updates resulting from a single quantifier instance must emulate those of the whole set of equivalent instances it stands in for.

Our model allows equivalent instantiations, so this requirement could be ignored without sacrificing completeness or conservativeness. We nonetheless reflect it in the design of the generation-tracking solver, through the definition of tracked generation:

Definition 11 (Tracked generation). *The generation for term t used by T is*

$$\mathrm{gen}^{\mathsf{T}}_{E,g}(t) := \max\{\min\{g(c) \mid \mathit{congruent}_E(c, s)\} \mid s \in t\}$$

Definition 12 (Enabled instances for T).

$$\varepsilon^{\mathsf{T}}_E(g) := \{q[\![\theta]\!] \mid w(q) + \mathrm{gen}^{\mathsf{T}}_{E,g}(\theta) \leq \mathsf{maxG}, q \in Q, \theta \textit{ is a term}\}$$

Definition 13 (Generation updates). *A quantifier instance $q[\![\theta]\!]$ causes g to be updated as follows:*

$$\mathrm{update}^{\mathsf{T}}_E(g, q[\![\theta]\!]) := g[t \mapsto \min(g(t), w(q) + \mathrm{gen}^{\mathsf{T}}_{E,g}(\theta)) \mid t \in q[\![\theta]\!]]$$

Generation updates are sticky, meaning we do not undo them when we pop instances. Doing so would not only require additional backtracking state, but also make the solver re-discover the same updates later, which is expensive in practice.

Modified transitions and definitions

$$\frac{q[\![\theta]\!] \in \varepsilon^{\mathsf{T}}_E(g)}{\langle \tau, g\rangle \to_{\mathsf{T}} \langle \tau \cdot \mathtt{inst}(q[\![\theta]\!]), \mathrm{update}^{\mathsf{T}}_E(g, q[\![\theta]\!])\rangle}\ (\text{PushInst}_{\mathsf{T}})$$

All other transitions remain unchanged except state augmentation with g.

Theorem 4. $\mathsf{N} \preceq \mathsf{T}$.

Proof. Simulating PushEq, PopEq, and PopInst is trivial (see the proof of Theorem 2). Lemma 6 in the appendix shows that $\text{PushInst}_{\mathsf{N}}$ can be simulated by a sequence of $\text{PushInst}_{\mathsf{T}}$ steps.

Theorem 5. $\mathsf{T} \preceq \mathsf{Id}$.

Proof. Since $\text{PushEq}_{\mathsf{T}}$, $\text{PopEq}_{\mathsf{T}}$, and $\text{PopInst}_{\mathsf{T}}$ are unchanged, simulating them remains trivial. By Lemma 8 in the appendix, there exists an E' such that $\forall t.\ \mathrm{gen}^{\mathsf{ideal}}_{E'}(t) \leq \mathrm{gen}^{\mathsf{T}}_{E(\tau),g}(t)$. A $\text{PushInst}_{\mathsf{T}}$ step can be simulated by first pushing all the equalities in E', then taking a PushInst step.

6.3 Refinement 3: Congruence-Class-Tracking Solver – C

T is still inefficient because computing $\min\{g(c) \mid \mathit{congruent}_E(c,t)\}$ is expensive for large congruence classes (§8 shows this empirically). The next version avoids iterating over congruence classes at match time by maintaining one generation number per congruence class.

Solver state is now a tuple $\Sigma_{\mathsf{C}} : \langle \tau, \kappa, \gamma, \delta \rangle$ where

- E, I have the same meaning as in Σ_{N} and Σ_{T}
- κ : Term $\to$ Set(Term) - maps a term to its congruence class
- γ : Set(Term) $\to \mathbb{N}$ - maps a congruence class to its generation number.
- We replace $\mathtt{eq}(e)$ by $\mathtt{eq}(e, \kappa, \gamma)$ to model backtracking to a snapshot
- δ : Set(Term $\times \mathbb{N}$) - cumulative set of sticky generation updates

Initially,

- $\delta = \emptyset$
- $\kappa(t) = \{t\}$, for all t.
- $\gamma(\{t\}) = 0$ if $t \in A$ and $\gamma(\cdot) = \infty$ otherwise, for all t.

C needs to explicitly keep track of congruence classes, which SMT solvers readily do. Congruence classes are merged, modifying κ, as a result of PushEq and unmerged as a result of PopEq. We make sure γ is updated accordingly, such that it always maps to the generation of the minimal term in the class.

Definition 14 (Merging congruence classes). *To track merging of congruence classes and updates to generations we introduce the relation* $\langle \kappa, \gamma \rangle \to_{CGC(E)} \langle \kappa', \gamma' \rangle$. *It can be obtained from the reflexive-transitive closure of the inference:*

$$\frac{\mathit{congruent}_E(t_1, t_2) \qquad X \in \kappa(t_1), Y \in \kappa(t_2) \qquad \kappa' = \kappa[t \mapsto X \cup Y \mid t \in X \cup Y]}{\langle \kappa, \gamma \rangle \to_{CGC(E)} \langle \kappa', \gamma[X \cup Y \mapsto \min(\gamma(X), \gamma(Y))] \rangle}$$

Definition 15 (Congruence-class-tracked generation). *The effective generation of a term t for* C *is:*

$$\mathrm{gen}^{\mathsf{C}}_{\kappa,\gamma}(t) := \max\{\gamma(\kappa(s)) \mid s \in t\}$$

γ is analogous to g in T. It is therefore updated similarly, in response to quantifier instantiations:

Definition 16 (Generation updates for C).

$$\mathrm{update}^{\mathsf{C}}_{\kappa}(\gamma, q[\![\theta]\!]) := \gamma[\kappa(t) \mapsto \min(\gamma(\kappa(t)), w(q) + \mathrm{gen}^{\mathsf{C}}_{\kappa,\gamma}(\theta)) \mid t \in q[\![\theta]\!]]$$

One complication that arises is that naively backtracking congruence-class merges may undo generation updates resulting from quantifier instantiation, which are meant to be sticky. It is crucial to prevent this, as re-discovering lower generations for existing terms is expensive in practice. To this end, we explicitly store a cumulative set of generation updates, which are re-applied on PopEq.

Definition 17 (New sticky updates caused by an instantiation). *Each term t re-discovered at a lower generation through a quantifier instance $q[\![\theta]\!]$ is tagged with a sticky update in $\Delta_{q[\![\theta]\!],\kappa}(\gamma,\gamma')$:*

$$\Delta_{q[\![\theta]\!],\kappa}(\gamma,\gamma') := \{\langle t, \gamma'(\kappa(t))\rangle \mid t \in q[\![\theta]\!] \wedge \gamma'(\kappa(t)) < \gamma(\kappa(t)) < \infty\}$$

Definition 18 (Replaying sticky updates). *Applying the updates stored in δ to $\langle \tau, \kappa, \gamma, \delta\rangle$ leads to $\langle \tau, \kappa, \gamma', \delta\rangle$ if $\gamma \to_{replay_\kappa(\delta)} \gamma'$ can be obtained from one of the inferences:*

$$\frac{}{\gamma \to_{replay_\kappa(\emptyset)} \gamma} \textit{ (ApplyNone)}$$

$$\frac{\gamma \to_{replay_\kappa(\delta)} \gamma'}{\gamma \to_{replay_\kappa(\delta \cup \{\langle t,n\rangle\})} \gamma'[(\kappa(t)) \mapsto \min(\gamma'(\kappa(t)), n)]} \textit{ (ApplySome)}$$

Definition 19 (Enabled instances for C).

$$\varepsilon^{\mathsf{C}}_{\kappa}(\gamma) := \{q[\![\theta]\!] \mid w(q) + \mathrm{gen}^{\mathsf{C}}_{\kappa,\gamma}(\theta) \leq \mathsf{maxG}, q \in Q, \theta \textit{ is a term}\}$$

We now present the transition rules for C:

$$\frac{e := (t_1 \simeq t_2) \qquad \langle \kappa, \gamma\rangle \to_{CGC(E \cup \{e\})} \langle \kappa', \gamma'\rangle}{\langle \tau, \kappa, \gamma, \delta\rangle \to_{\mathsf{C}} \langle \tau \cdot \mathtt{eq}(e, \kappa, \gamma), \kappa', \gamma', \delta\rangle} \text{ (PushEq } \mathsf{C})$$

$$\frac{\gamma' \to_{\mathrm{replay}_\kappa(\delta)} \gamma''}{\langle \tau \cdot \mathtt{eq}(e, \kappa', \gamma'), \kappa, \gamma, \delta\rangle \to_{\mathsf{C}} \langle \tau, \kappa', \gamma'', \delta\rangle} \text{ (PopEq } \mathsf{C})$$

$$\frac{q[\![\theta]\!] \in \varepsilon^{\mathsf{C}}_{\kappa}(\gamma) \qquad \gamma' = \mathrm{update}^{\mathsf{C}}_{\kappa}(\gamma, q[\![\theta]\!])}{\langle \tau, \kappa, \gamma, \delta\rangle \to_{\mathsf{C}} \langle \tau \cdot \mathtt{inst}(q[\![\theta]\!]), \kappa, \gamma', \delta \cup \Delta_{q[\![\theta]\!],\kappa}(\gamma, \gamma')\rangle} \text{ (PushInst } \mathsf{C})$$

$$\frac{}{\langle \tau \cdot \mathtt{inst}(q[\![\theta]\!]), \kappa, \gamma, \delta\rangle \to_{\mathsf{C}} \langle \tau, \kappa, \gamma, \delta\rangle} \text{ (PopInst } \mathsf{C})$$

Lemma 3. *Every transition sequence executable from the initial state of C is executable from the initial state of T, and vice versa; moreover, the resulting states satisfy*

$$\forall t.\ \gamma(\kappa(t)) = \min\{g(c) \mid \textit{congruent}_E(c,t)\}$$

Proof. By induction on the length of the transition sequence. The invariant holds initially, since both maps assign generation 0 to input terms and ∞ to all other terms.

On PushEq, merging congruence classes updates γ to the minimum of the two merged classes' generations. This is the same as $\min\{g(c) \mid \textit{congruent}_E(c,t)\}$ once the classes are merged.

On PushInst, C updates $\gamma(\kappa(t))$ for every subterm t in the instance using the same weight and, by the induction hypothesis, the same generation for θ as T uses to update g.

On PopEq C, symmetrically to PushEq C, unmerging congruence classes restores the minimum over the resulting classes. The sticky-update mechanism ensures no generation updates are lost in the process: any update to a term's generation, once recorded as a sticky update, survives the split and is re-applied.

PopInst C does not affect either map. So the invariant is trivially preserved.

Theorem 6. $\mathsf{T} \preceq \mathsf{C}$.

Proof. Given σ_{T} and σ_{C}, the latter can simulate a step to σ'_{T} by first replaying the transition sequence $\sigma_{\mathsf{T}}^{init} \rightsquigarrow_{\mathsf{T}} \sigma_{\mathsf{T}}$ and then taking the same final step. The sequence can be replayed due to Lemma 3, and the fact that the value of $\gamma(\kappa(t))$ at σ_{C} is no greater than its value at $\sigma_{\mathsf{C}}^{init}$ for all t.

Theorem 7. $\mathsf{C} \preceq \mathsf{T}$.

Proof. The proof is symmetric to that of Theorem 6.

7 Z3 Implementation

We implemented C in Z3; our implementation has been upstream since version 5.0. This section briefly describes some of the implementation details.

Z3 associates each internalized term with a metadata structure called an e-node. It keeps track of congruence classes through a hash table which stores a unique e-node per congruence class, called the *congruence representative*. Z3 uses this table during pattern matching to ensure it only binds congruence representatives, in order to avoid duplicate matches. This table effectively stores κ from C.

One option, in line with T, would be to store the function g as an e-node field set at internalization and updated during quantifier instantiation. At matching time, we could walk the congruence class of each subterm of the match candidate to determine the minimum generation. We experimented with this option but discarded it, as it led to significant performance regression (see §8).

Another option, in line with C, would be to forego g and instead store γ, directly in the hash table entry. This performs much better than the prior option, but still incurs a minor (5%) slowdown, since looking up the hash table is less cache-friendly than reading the generation number directly off the e-node.

Our implementation instead stores γ on the e-node of the congruence root. When congruence classes get merged, we update the new root's e-node and save the old generation on the backtracking stack. This wastes some space, since the generation field goes unused on every e-node that is not a congruence root, but the tradeoff is worthwhile: in the common case the congruence root's e-node is already accessed before a generation lookup, so we avoid the extra access.

We store δ explicitly as a list of sticky updates to be re-applied whenever congruence classes get unmerged. This can be expensive: sticky updates survive backtracking until the term whose generation was updated is garbage collected,

and re-applying all of them costs time linear in the number of updates. In practice, however, discovering an existing term at a lower generation is uncommon enough (despite its significance for instability) that this cost is negligible.

Propagating generation updates is another potentially expensive consequence of our implementation. When a term whose generation previously dominated other terms' generations (via instantiation) is itself updated, that update must propagate, e.g., by allowing duplicate matches with lower generation numbers. Our implementation skips this, slightly deviating from C, yet still greatly improves stability. We suspect enabling such duplicate matches would not significantly affect performance for the same reason as above: generation updates are rare in practice.

8 Empirical Evaluation

Experimental setup: For our evaluation, we use the Mariposa [16] framework, and its unstable-core benchmark set. This consists of 60 queries from prior program verification projects, mostly written in Dafny [8]. Mariposa stability-fuzzes these queries by generating mutants through reseeding, assertion shuffling and variable renaming.

For each query, we generate 99 mutants and run each of them, along with the original query, with a 60-second timeout (Mariposa's default). We include queries that are not outcome-unstable to ensure we do not introduce regressions that increase timeouts. Our experiments were run on a machine with two 64-core AMD EPYC 7763 CPUs at 2.4GHz and 503GiB of RAM. For two of the queries, running all 100 variants in parallel allocates enough memory to crash our infrastructure; three others produce mutants that return a solver error due to malformed input. We discard these five queries and present results for the remaining 55 queries, producing 5500 mutants, each of which returns `unsat`, `unknown`, or `timeout`.

We compare four versions of generation accounting: z3's accounting prior to our work, which we treat as a baseline, the two less efficient options outlined in §7 (`z3-T`, `z3-C-table`), and our final implementation (`z3-C-enode`).

Solver version	Outcome-unstable queries	Unknown mutants	Timeouts
`baseline`	27	489	1813
`z3-T`	6	25	2399
`z3-C-table`	4	25	1867
`z3-C-enode`	4	29	1779

Table 1. Stability and performance results on Mariposa's unsat core benchmark across implementation versions. Experiments include 55 queries generating 5500 mutants.

Table 1 reports, for each version, the number of outcome-unstable queries (i.e., queries that have at least one `unknown` and one `unsat` mutant), the total number of `unknown` mutants across all queries, and the total number of timeouts.

Stability: All three implementations substantially reduce outcome instability relative to the baseline: `z3-C-enode` cuts the number of unknown mutants by 94% and the number of outcome-unstable queries by 85%, with `z3-T` and `z3-C-table` achieving comparable reductions; the differences are within noise.

Performance: We also observe a performance cost, measured as the total number of timeouts across all queries: `z3-T` is considerably slower, producing 32% more timeouts than the baseline, while `z3-C-table` incurs only a minor regression (3% more timeouts). `z3-C-enode`, our final implementation, avoids this cost and yields slightly fewer timeouts than the baseline. Additionally, for each solver version, we manually examined the timeout count per query and confirmed that our changes did not cause any single query to blow up.

To ensure our changes do not introduce regressions beyond program verification, we also ran `z3-C-table` and `z3-C-enode` on the non-incremental SMT-LIB benchmarks [13], using a 20-second timeout. For quantified categories, all differences fall within noise, the largest being 14 additional timeouts (1%) for `z3-C-table` on ALIA. Among quantifier-free categories, the only notable difference is a 5% slowdown for `z3-C-table` on QF_UF, which does not cause additional timeouts. This is expected: the same cache-locality issue affecting Mariposa performance also slows down congruence closure in the absence of quantifiers. `z3-C-enode` avoids this regression entirely.

References

1. Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength SMT solver. In: TACAS (2022)
2. Bjørner, N., Eisenhofer, C., Gurfinkel, A., Lopes, N.P., de Moura, L., Nachmanson, L., Wintersteiger, C.: Z3 internals. https://github.com/Z3Prover/doc/blob/master/internals/z3internals.mdk
3. Buchberger, B.: Gröbner bases: An algorithmic method in polynomial ideal theory. In: Bose, N.K. (ed.) Multidimensional Systems Theory, pp. 184–232. D. Reidel Publishing Company (1985). https://doi.org/10.1007/978-94-009-5225-6_6
4. Cebeci, C., Bjørner, N., Candea, G., Pit-Claudel, C.: A conjecture regarding SMT instability (2025)
5. Church, A., Rosser, J.B.: Some properties of conversion. Transactions of the American Mathematical Society **39**(3), 472–482 (1936)
6. Knuth, D.E., Bendix, P.B.: Simple word problems in universal algebras. In: Computational Problems in Abstract Algebra, pp. 263–297 (1970)
7. Lattuada, A., Hance, T., Bosamiya, J., Brun, M., Cho, C., LeBlanc, H., Srinivasan, P., Achermann, R., Chajed, T., Hawblitzel, C., Howell, J., Lorch, J.R., Padon, O., Parno, B.: Verus: A practical foundation for systems verification. In: SOSP (2024)
8. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: LPAR (2010)

9. Leino, K.R.M., Pit-Claudel, C.: Trigger selection strategies to stabilize program verifiers. In: CAV (2016)
10. de Moura, L., Bjørner, N.: Efficient e-matching for smt solvers. In: CADE (2007)
11. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS (2008)
12. Nieuwenhuis, R., Oliveras, A.: Fast congruence closure and extensions. In: Information and Computation (2007)
13. Preiner, M., Schurr, H.J., Barrett, C., Fontaine, P., Niemetz, A., Tinelli, C.: SMT-LIB release 2025 (non-incremental benchmarks), https://doi.org/10.5281/zenodo.16740866
14. Swamy, N., Hritcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P.Y., Kohlweiss, M., Zinzindohoue, J.K., Zanella-Béguelin, S.: Dependent types and multi-monadic effects in F*. In: POPL (2016)
15. Zhou, Y., Bosamiya, J., Li, J., Heule, M.J., Parno, B.: Context pruning for more robust SMT-based program verification. In: FMCAD (2024)
16. Zhou, Y., Bosamiya, J., Takashima, Y., Li, J., Heule, M., Parno, B.: Mariposa: Measuring SMT instability in automated program verification. In: FMCAD (2023)
17. Zhou, Y., Shah, A., Lin, Z., Heule, M., Parno, B.: Cazamariposas: Automated instability debugging in SMT-based program verification. In: CADE (2025)

Appendix A. Proofs related to N

.

Lemma 4 (N can find all ideal generations within threshold).

$$\mathrm{gen}_E^{\mathsf{ideal}}(t) \leq \mathsf{maxG} \implies \exists \tau'.\langle\tau\rangle \rightsquigarrow_{\mathsf{N}}^{inst} \langle\tau'\rangle \wedge \tau \sqsubseteq \tau' \wedge \mathrm{gen}_{E,I(\tau')}(t) \leq \mathrm{gen}_E^{\mathsf{ideal}}(t)$$

where $\rightsquigarrow_{\mathsf{N}}^{inst}$ denotes reachability through exclusively PushInst$_{\mathsf{N}}$ transitions.

Proof. Since $\mathrm{gen}_E^{\mathsf{ideal}}(\theta) \leq G < \infty$, we have an inference for $t \vdash_{E,I^\infty} \mathrm{gen}_E^{\mathsf{ideal}}(t)$. By induction on this inference, we have three cases:

- (*Input*) $t \in A$: Then $\mathrm{gen}_{E,I}(t) = 0 = \mathrm{gen}_E^{\mathsf{ideal}}(t)$. So $\tau' := \tau$ is a witness.
- (*Congruence*) $t = t'[s_2], s_1 \cong_E s_2, t'[s_1] \vdash_{E,I^\infty} n_{t'}, s_2 \vdash_{E,I^\infty} n_{s_2}, \mathrm{gen}_E^{\mathsf{ideal}}(t) = \max(n_{t'}, n_{s_2})$: We have $n_{t'} \leq \mathsf{maxG}$ and $n_{s_2} \leq \mathsf{maxG}$ and so, by the induction hypotheses, we get $\langle\tau\rangle \rightsquigarrow_{\mathsf{N}}^{\mathrm{inst}} \langle\tau_1\rangle \wedge \mathrm{gen}_{E,I(\tau_1)}(t[s_1]) \leq n_{t'}$ and $\langle\tau\rangle \rightsquigarrow_{\mathsf{N}}^{\mathrm{inst}} \langle\tau_2\rangle \wedge \mathrm{gen}_{E,I(\tau_2)}(s_2) \leq n_{s_2}$. Since adding instances never blocks PushInst$_{\mathsf{N}}$, we can replay the transitions in $\langle\tau\rangle \rightsquigarrow_{\mathsf{N}}^{\mathrm{inst}} \langle\tau_2\rangle$ from $\langle\tau_1\rangle$ and get a τ' with $\langle\tau\rangle \rightsquigarrow_{\mathsf{N}}^{\mathrm{inst}} \langle\tau_1\rangle \rightsquigarrow_{\mathsf{N}}^{\mathrm{inst}} \langle\tau'\rangle$, $\tau_1 \sqsubseteq \tau'$, and $\tau_2 \sqsubseteq \tau'$. Then $\mathrm{gen}_{E,I(\tau')}(t[s_1]) \leq \mathrm{gen}_{E,I(\tau_1)}(t[s_1]) \leq n_{t'}$ and similarly $\mathrm{gen}_{E,I(\tau')}(s_2) \leq n_{s_2}$. So, through the *Congruence* rule, $\mathrm{gen}_{E,I(\tau')}(t) \leq n = \mathrm{gen}_E^{\mathsf{ideal}}(t)$.
- (*Inst*) $t \in q[\![\theta]\!], \mathrm{gen}_E^{\mathsf{ideal}}(t) = w(q) + n_\theta, \theta \vdash_{E,I^\infty} n_\theta$: Since quantifier weights are non-negative, $n_\theta \leq n \leq \mathsf{maxG}$. Then by the induction hypothesis we have a τ'' with $\mathrm{gen}_{E,I(\tau'')}(\theta) \leq \mathrm{gen}_E^{\mathsf{ideal}}(\theta) \wedge \langle\tau\rangle \rightsquigarrow_{\mathsf{N}}^{\mathrm{inst}} \langle\tau'\rangle \wedge \tau \sqsubseteq \tau'$. PushInst$_{\mathsf{N}}$ with $q[\![\theta]\!]$ is enabled in $\langle\tau''\rangle$ and leads to some $\langle\tau'\rangle$ such that τ' is a witness.

Lemma 5 (The PushInst case of Theorem 2). *For all* $\sigma_{\mathsf{N}}, \sigma_{\mathsf{Id}}, \sigma'_{\mathsf{Id}}$,

$$\mathcal{T}(\sigma_{\mathsf{Id}}) \sqsubseteq \mathcal{T}(\sigma_{\mathsf{N}}) \wedge \sigma_{\mathsf{Id}} \rightarrow^{inst}_{\mathsf{Id}} \sigma'_{\mathsf{Id}} \implies \exists \sigma'_{\mathsf{N}}.\ \sigma_{\mathsf{N}} \rightsquigarrow_{\mathsf{N}} \sigma'_{\mathsf{N}} \wedge \mathcal{T}(\sigma'_{\mathsf{Id}}) \sqsubseteq \mathcal{T}(\sigma'_{\mathsf{N}})$$

Proof. Let E be $E(\mathcal{T}(\sigma_{\mathsf{Id}}))$ (or equivalently $E(\mathcal{T}(\sigma_{\mathsf{N}}))$, since $\mathcal{T}(\sigma_{\mathsf{Id}}) \sqsubseteq \mathcal{T}(\sigma_{\mathsf{N}})$). From $\sigma_{\mathsf{Id}} \rightarrow^{\text{inst}}_{\mathsf{Id}} \sigma'_{\mathsf{Id}}$ we have $\text{gen}^{\mathsf{ideal}}_E(\theta) + w(q) \leq \mathsf{maxG}$, where $q[\![\theta]\!]$ is the instance being pushed. By Lemma 4, we have a τ' with $\text{gen}_{E,I(\tau')}(\theta) \leq \text{gen}^{\mathsf{ideal}}_E(\theta) \wedge \sigma_{\mathsf{N}} \rightsquigarrow^{\text{inst}}_{\mathsf{N}} \langle \tau' \rangle \wedge \mathcal{T}(\sigma_{\mathsf{N}}) \sqsubseteq \tau'$ $\text{PushInst}_{\mathsf{N}}$ with $q[\![\theta]\!]$ is enabled in $\langle \tau' \rangle$ and leads to some σ'_{N} with $\mathcal{T}(\sigma'_{\mathsf{Id}}) \sqsubseteq \mathcal{T}(\sigma'_{\mathsf{N}})$.

Appendix B. Proofs related to T

Lemma 6 (T can find derivable generations). *For all* t, n, τ, g *such that* $\sigma^{init}_{\mathsf{T}} \rightsquigarrow_{\mathsf{T}} \langle \tau, g \rangle$,

$$t \vdash_{E,I} n \implies \exists \tau', g'.\ \langle \tau, g \rangle \rightsquigarrow^{inst}_{\mathsf{T}} \langle \tau', g' \rangle \wedge \text{gen}^{\mathsf{T}}_{E,g'}(t) \leq n$$

Proof. By induction on $t \vdash_{E,I} n$, we have three cases:

- (*Input*) $t \in A, n = 0$: Since A is subterm-closed, $t' \in A$ for all subterms t' of t. Then we have $g^{init}(t') = 0$ at initialization and because $g(t')$ cannot increase through any transition, $g(t') = 0$. So $\tau' := \tau, g' := g$ is a witness.
- (*Inst*) $t \in q[\![\theta]\!], \text{gen}_{E,I}(t) = w(q) + n_\theta, \theta \vdash_{E,I} n_\theta$: By the induction hypothesis we have $\langle \tau, g \rangle \rightsquigarrow^{\text{inst}}_{\mathsf{T}} \langle \tau'', g'' \rangle$ and $\text{gen}^{\mathsf{T}}_{E,g''}(t) \leq n_\theta$. $\text{PushInst}_{\mathsf{T}}$ is enabled at $\langle \tau'', g'' \rangle$ and leads to some $\langle \tau', g' \rangle$ that serves as a witness.
- (*Congruence*) $t = t'[s_2], s_1 \cong_E s_2, t'[s_1] \vdash_{E,I} n_{t'}, s_2 \vdash_{E,I} n_{s_2}, \text{gen}_{E,I}(t) = \max(n_{t'}, n_{s_2})$: Similarly to the argument in the proof of Lemma 4, we can take the transitions described by the two inductive hypotheses to get a τ', g' with $\langle \tau, g \rangle \rightsquigarrow^{\text{inst}}_{\mathsf{T}} \langle \tau', g' \rangle$, $\text{gen}^{\mathsf{T}}_{E,g'}(t'[s_1]) \leq n_{t'}$, and $\text{gen}^{\mathsf{T}}_{E,g'}(s_2) \leq n_{s_2}$. It suffices to show $\min\{g'(c) \mid \mathit{congruent}_E(c, t_s)\} \leq \text{gen}_{E,I}(t)$ for all subterms t_s of $t'[s_2]$. We consider three cases:
  - $t_s \in s_2$: Since $\text{gen}^{\mathsf{T}}_{E,g''}(s_2) \leq n_{s_2}$, $\min\{g'(c) \mid \mathit{congruent}_E(c, t_s)\} \leq n_{s_2} \leq \text{gen}_{E,I}(t)$.
  - $t_s \in t'[s_1]$: The same argument applies with $\text{gen}^{\mathsf{T}}_{E,g''}(t'[s_1]) \leq n_{t'}$
  - $t_s = t''[s_2], t''[s_1] \in t'[s_1]$: We have $\min\{g'(c) \mid \mathit{congruent}_E(c, t''[s_1])\} \leq n_{t'}$ due to $\text{gen}^{\mathsf{T}}_{E,g''}(t'[s_1]) \leq n_{t'}$. Since $\mathit{congruent}_E(t''[s_1], t''[s_2])$, the same minimal c ensures $\min\{g'(c) \mid \mathit{congruent}_E(c, t''[s_2])\} \leq n_{t'} \leq \text{gen}_{E,I}(t)$.

Lemma 7 (Mapped values for subterms). *For all reachable states* $\langle \tau, g \rangle$ *of* T*, for all terms* t, s, $s \in t \implies g(s) \leq g(t)$.

Proof. Since A is subterm-closed, initially either $g(t) = g(s) = 0$ or $g(t) = g(s) = \infty$. Then, $g(t)$ and $g(s)$ may only decrease, via $\text{PushInst}_{\mathsf{T}}$ transitions, which never decrease $g(t)$ further than $g(s)$.

Lemma 8 (Tracked generations are lower-bounded by some ideal). *For all reachable states $\langle \tau, g \rangle$ of T,*

$$\exists E'. \forall t.\ \mathrm{gen}^{\mathsf{ideal}}_{E'}(t) \leq \mathrm{gen}^{\mathsf{T}}_{E(\tau),g}(t)$$

Proof. By induction on the reachability of $\langle \tau, g \rangle$.

- Initially, τ is empty. We choose $E' := \emptyset$. Then for all $t \in A$, both generations are 0 ($\mathrm{gen}^{\mathsf{T}}_{E(\tau),g}(t) = 0$ because A is subterm-closed), and for $t \notin A$ both generations are ∞.
- (PushEq) $\tau := \tau' \cdot \mathtt{eq}(e)$, $\mathrm{IH}_1 := \forall t.\ \mathrm{gen}^{\mathsf{ideal}}_{E'}(t) \leq \mathrm{gen}^{\mathsf{T}}_{E(\tau'),g}(t)$: We choose the witness $E'' := E' \cup E(\tau)$. Proof by structural induction on t. Let $n := \mathrm{gen}^{\mathsf{T}}_{E(\tau),g}(t)$.
  - If t is a constant (0-arity function), the only term congruent to t is t itself. Then $g(t) = n$ and $\mathrm{gen}^{\mathsf{T}}_{E(\tau'),g}(t) = n$. By IH_1, $\mathrm{gen}^{\mathsf{ideal}}_{E'}(t) \leq n$. Since $E' \subseteq E''$, $\mathrm{gen}^{\mathsf{ideal}}_{E''}(t) \leq n$.
  - Otherwise $t := c[a]$. By the structural induction hypothesis, $\mathrm{gen}^{\mathsf{ideal}}_{E''}(a) \leq n$. Assume all terms congruent to t can be expressed as $c[b]$ for some b with $a \cong_{E(\tau)} b$. In reality, a congruent term can be obtained by rewriting more than one argument of the top-level function as well, but we ignore this for simplicity of presentation. For some b, $g(c[b]) \leq n$. Then, by Lemma 7, $\mathrm{gen}^{\mathsf{T}}_{E(\tau'),g}(c[b]) \leq n$ and so by IH_1, $\mathrm{gen}^{\mathsf{ideal}}_{E'}(c[b]) \leq n$. Since $E' \subseteq E''$, $\mathrm{gen}^{\mathsf{ideal}}_{E''}(c[b]) \leq n$ and since $E(\tau) \subseteq E'$, $a \cong_{E''} b$. Then by the *Congruence* rule for deriving generations, $\mathrm{gen}^{\mathsf{ideal}}_{E''}(c[a]) \leq n$.
- (PopEq) $\tau \cdot \mathtt{eq}(e) = \tau'$, $\mathrm{IH}_1 := \forall t.\ \mathrm{gen}^{\mathsf{ideal}}_{E'}(t) \leq \mathrm{gen}^{\mathsf{T}}_{E(\tau'),g}(t)$: Since $E(\tau) \subseteq E(\tau')$, for all t, $\mathrm{gen}^{\mathsf{T}}_{E(\tau'),g}(t) \leq \mathrm{gen}^{\mathsf{T}}_{E(\tau),g}(t)$. Then $\mathrm{gen}^{\mathsf{ideal}}_{E'}(t) \leq \mathrm{gen}^{\mathsf{T}}_{E(\tau),g}(t)$ so E' serves as a witness.
- (PushInst) $\tau = \tau' \cdot \mathtt{inst}(q[\![\theta]\!]), g = \mathrm{update}^{\mathsf{T}}_{E}(g', q[\![\theta]\!]), \mathrm{IH}_1 := \forall t.\ \mathrm{gen}^{\mathsf{ideal}}_{E'}(t) \leq \mathrm{gen}^{\mathsf{T}}_{E(\tau'),g'}(t)$: We choose $E'' := E' \cup E(\tau)$ as the witness. The proof is structurally inductive, like the PushEq case. Let $n := \mathrm{gen}^{\mathsf{T}}_{E(\tau),g}(t)$.
  - If t is a constant, $g(t) = n$ and $g'(t) = \mathrm{gen}^{\mathsf{T}}_{E(\tau'),g'}(t)$. Since $E' \subseteq E''$, $\mathrm{gen}^{\mathsf{ideal}}_{E''}(t) \leq n$. If $g(t) = g'(t)$ we conclude by IH_1. Otherwise, $g(t) = w(q) + \mathrm{gen}^{\mathsf{T}}_{E,g'}(\theta)$ with $t \in q[\![\theta]\!]$. Since $E(\tau) \subseteq E''$, $w(q) + \mathrm{gen}^{\mathsf{ideal}}_{E''}(\theta) \leq n$. Then by the *Inst* rule for deriving generations, $\mathrm{gen}^{\mathsf{ideal}}_{E''}(t) \leq n$.
  - If t is a function application, we follow an identical argument to the second case of (PushEq). An ideal generation less than n can be derived for $c[b]$ via the *Inst* rule, and then for $c[a]$ via the *Congruence* rule.
- (PopInst): $E(\tau) = E(\tau')$ and $g = g'$, so we conclude trivially via the induction hypothesis.