Logical Methods in Computer Science
Volume 19, Issue 4, 2023, pp. 22:1–22:27
https://lmcs.episciences.org/

Submitted Aug. 30, 2022
Published Dec. 08, 2023

CONSTRUCTIVE MANY-ONE REDUCTION FROM THE HALTING PROBLEM TO SEMI-UNIFICATION (EXTENDED VERSION*)

ANDREJ DUDENHEFNER

TU Dortmund University, Dortmund, Germany
e-mail address: andrej.dudenhefner@cs.tu-dortmund.de

Abstract. Semi-unification is the combination of first-order unification and first-order matching. The undecidability of semi-unification has been proven by Kfoury, Tiuryn, and Urzyczyn in the 1990s by Turing reduction from Turing machine immortality (existence of a diverging configuration). The particular Turing reduction is intricate, uses non-computational principles, and involves various intermediate models of computation.

The present work gives a constructive many-one reduction from the Turing machine halting problem to semi-unification. This establishes RE-completeness of semi-unification under many-one reductions. Computability of the reduction function, constructivity of the argument, and correctness of the argument is witnessed by an axiom-free mechanization in the Coq proof assistant. Arguably, this serves as comprehensive, precise, and surveyable evidence for the result at hand. The mechanization is incorporated into the existing, well-maintained Coq library of undecidability proofs. Notably, a variant of Hooper's argument for the undecidability of Turing machine immortality is part of the mechanization.

1. Introduction

Semi-unification is the following decision problem: given a finite set of pairs of first-order terms, is there a substitution φ such that for each pair (σ, τ) in the set of first-order terms we have $\psi(\varphi(\sigma)) = \varphi(\tau)$ for some substitution ψ?

Semi-unification was formulated independently by Henglein [Hen88], by Leiß [Lei89b], and by Kapur, Musser, Narendran and Stillman [KMNS88]. Both Henglein [Hen88] and Leiß [Lei89b] show that semi-unification is equivalent with type inference for functional programming languages with recursive polymorphism[1] [Myc84] (for a detailed account

Key words and phrases: constructive mathematics, computability theory, undecidability, semi-unification, mechanization, Coq.

* The present work is an extended version of a prior conference paper [Dud22b].

The author is grateful for encouragement and assistance by Paweł Urzyczyn and the members of the programming systems lab led by Gert Smolka at Saarland University. Additionally, the author thanks the anonymous reviewers for insightful remarks on the manuscript and on the intertwined history of semi-unification and polymorphic type inference.

[1] Mycroft's original type inference algorithm for ML with recursive polymorphism predates the conception of semi-unification and is based on Algorithm W by Damas and Milner [DM82, Section 6].

DOI:10.46298/LMCS-19(4:22)2023

see [Lei89a, Hen93, KTU93a]). Intuitively, the substitution φ establishes global code invariants (cf. first-order unification), and the individual substitutions ψ establish additional local conditions for each polymorphic function application (cf. first-order matching).

While both first-order unification and first-order matching are decidable problems, the status of semi-unification remained open, until answered negatively by Kfoury, Tiuryn, and Urzyczyn [KTU90b, KTU93b]. The undecidability of semi-unification impacted programming language design and analysis with respect to polymorphic recursion [LH91, JK93], loop detection [Pur87], and data flow [FRD00]. Another prominent result based on the undecidability of semi-unification is the undecidability of System F [Gir72, Rey74] typability and type checking [Wel99, Dud21]. Of course, the negative result motivated the complementary line of work [LH91] in search for expressive, decidable fragments of semi-unification. A notable decidable fragment is *acyclic* semi-unification [KTU94], used for standard ML typability[2].

Due to the importance of semi-unification in functional programming, it is natural to ask for *surveyable* evidence (both locally and globally in the sense of [Bas06]) for its undecidability. The original undecidability proof [KTU90b] is quite sophisticated, and it was simplified [Dud20] and partially mechanized in the Coq proof assistant. Another peculiarity of semi-unification is that prior to the negative result, it was erroneously claimed to be *decidable* [Pur87, U-match Algorithm], [KTU88, Lemma 5.1], [KMNS88, Algorithm A-1]. This serves as further evidence for the intricate nature of semi-unification and calls for surveyable evidence. Unfortunately, there are several aspects that obstruct surveyability of previous work.

First, existing arguments rely on the undecidability of Turing machine immortality, shown by Hooper [Hoo66]. The corresponding construction has received more attention [KO08], however, it was never published in full detail. Hooper remarks:

> *A routine and unimaginative analysis-of-cases proof would point this out more clearly; but it has remained unwritten since, as a rather tedious insult to the alert, qualified reader, it would surely remain unread.*

While the omissions are justified by accessibility, they hinder verification in full detail. The existing mechanization [Dud20] of the undecidability of semi-unification does not improve upon this aspect, as it assumes the undecidability of Turing machine immortality as an axiom.

Second, existing arguments use non-constructive principles such as excluded middle, König's lemma [KTU90b], or the fan theorem [Dud20] that do not support constructivity of the arguments. As a result, anti-classical theories, such as synthetic computability theory [Bau05], may be in conflict with such constructions. The question arises, whether non-constructive principles are inherent to semi-unification or could be avoided.

Third, existing arguments use Turing reductions and are insufficient to establish RE-completeness of semi-unification under many-one reductions. Hitherto, a many-one reduction from Turing machine halting to semi-unification is not given.

This work improves upon the above aspects as follows. It provides a comprehensive chain of many-one reductions from Turing machine halting to semi-unification, replacing immortality with uniform boundedness. Crucially, each many-one reduction is mechanized in full detail using the Coq proof assistant [Coqa]. The mechanization witnesses correctness and

[2]EXPTIME-completeness of standard ML typability is shown by Kannellakis and Mitchell [KM89] (upper bound) and by Mairson, Kfoury, Tiuryn, and Urzyczyn [Mai90, KTU90a] (lower-bound).

constructivity of the argument. Specifically, the notion of a *constructive* proof is identified with an axiom-free Coq mechanization (cf. calculus of inductive constructions). It neither assumes functional extensionality[3], Markov's principle[4], nor the fan theorem[5]. Finally, the mechanization is integrated into the Coq Library of Undecidability Proofs [FLWD$^+$20], and contributes a (first of its kind) mechanized variant of Hooper's immortality construction [Hoo66].

The described improvements allow for an alternative approach to show many-one equivalence of System F typability and System F type checking, compared to the argument established in the 1990s by Wells [Wel99]. The original argument interreduces type checking and typability directly, which requires a technically sophisticated argument. Having a constructive many-one reduction from Turing machine halting to semi-unification at our disposal (together with recursive enumerability of System F typability and type checking), it suffices (and is simpler) to reduce semi-unification to type checking and typability individually [Dud21].

Synopsis. The reduction from Turing machine halting to semi-unification is divided into several reduction steps. Each reduction step is many-one and constructive. That is, a predicate P over the domain X *constructively many-one* reduces to a predicate Q over the domain Y, if there exists a computable function $f : X \to Y$ such that for all $x \in X$ we constructively have $P(x) \iff Q(f(x))$. Additionally, each reduction step is mechanized as part of the Coq Library of Undecidability Proofs [FLWD$^+$20].

Section 2: Turing machine halting is reduced to two-counter machine halting (Lemma 2.4) using Minsky's argument [Min67, Section 14.1]. This step simplifies the starting machine model.

Section 3: Two-counter machine halting is reduced to one-counter machine 1-halting (Lemma 3.7), adapting Minsky's observation [Min67, Section 14.2]. This step prepares the machine model for nested simulation via two-stack machines without symbol search.

Section 4.1: One-counter machine 1-halting is reduced to deterministic, length-preserving two-stack machine uniform boundedness (Lemma 4.16), adapting Hooper's construction for Turing machine immortality [Hoo66]. This step transitions the machine problem from halting to uniform boundedness.

Section 4.2: Deterministic, length-preserving two-stack machine uniform boundedness is reduced to confluent, simple two-stack machine uniform boundedness (Lemma 4.27), simplifying machine structure. This step enables reuse of the existing mechanized reduction from a uniform boundedness problem to semi-unification [Dud20].

Section 5.1: Confluent, simple two-stack machine uniform boundedness is reduced to simple semi-unification (Lemma 5.13), strengthening previous work [Dud20]. This step transitions to an undecidable fragment of semi-unification.

Section 5.2: Simple semi-unification is reduced to right-uniform, two-inequality semi-unification (Lemma 5.18), establishing the main result (Theorem 5.23).

Section 6: Outline of the mechanization of the above reduction steps.

Section 7: Concluding remarks.

[3]Two functions are equal if their values are equal at every argument (cf. homotopy type theory).

[4]If it is impossible that an algorithm does not terminate, then it will terminate (cf. Russian constructivism).

[5]The topological space of unbounded binary sequences is compact (cf. Brouwer's intuitionism).

2. Two-counter Machines

The key insight of recent work [Dud20] establishes a direct correspondence between semi-unification and a uniform boundedness problem for a machine model. In order to reduce Turing machine halting to such a problem, in this section we consider *two-counter machines* as a well-understood, mechanized [FL19], and more convenient intermediate model of computation.

Two-counter machines, pioneered by Minsky [Min67, Section 14.1], constitute a particularly simple, Turing-complete model of computation. A two-counter machine (Definition 2.1) stores data in two *counters*, each containing a natural number. A program instruction may either increment or decrement a counter value, and modify the current program index. The size of a two-counter machine is the length, denoted $|\cdot|$, of the list, denoted $[\ldots]$, of its instructions. Individual instructions in the list are indexed starting from index 0.

Definition 2.1 (Two-counter Machine ($\mathcal{M}$))**.** A *two-counter machine* $\mathcal{M}$ is a list of *instructions* of shape either $\mathtt{inc}_0$, $\mathtt{inc}_1$, $\mathtt{dec}_0\, j$, or $\mathtt{dec}_1\, j$, where $j \in \mathbb{N}$ is a *program index*.

A *configuration* of $\mathcal{M}$ is of shape $(i, (a, b))$, where $i \in \mathbb{N}$ is the current *program index* and $a, b \in \mathbb{N}$ are the current *counter values*.

The *step relation* of $\mathcal{M}$ on configurations, written $(\longrightarrow_{\mathcal{M}})$, is given by

- if $\mathtt{inc}_0$ is at index i in $\mathcal{M}$, then $(i, (a, b)) \longrightarrow_{\mathcal{M}} (i+1, (a+1, b))$;
- if $\mathtt{inc}_1$ is at index i in $\mathcal{M}$, then $(i, (a, b)) \longrightarrow_{\mathcal{M}} (i+1, (a, b+1))$;
- if $\mathtt{dec}_0\, j$ is at index i in $\mathcal{M}$, then $(i, (0, b)) \longrightarrow_{\mathcal{M}} (i+1, (0, b))$ and $(i, (a+1, b)) \longrightarrow_{\mathcal{M}} (j, (a, b))$;
- if $\mathtt{dec}_1\, j$ is at index i in $\mathcal{M}$, then $(i, (a, 0)) \longrightarrow_{\mathcal{M}} (i+1, (a, 0))$ and $(i, (a, b+1)) \longrightarrow_{\mathcal{M}} (j, (a, b))$;
- otherwise, we say that the configuration $(i, (a, b))$ is *halting*.

The *reachability relation* of $\mathcal{M}$ on configurations, written $(\longrightarrow^*_{\mathcal{M}})$, is the reflexive, transitive closure of $(\longrightarrow_{\mathcal{M}})$.

A configuration $(i, (a, b))$ *terminates* in $\mathcal{M}$, if we have $(i, (a, b)) \longrightarrow^*_{\mathcal{M}} (i', (a', b'))$ for some halting configuration $(i', (a', b'))$.

Remark 2.2. The above Definition 2.1 is proposed by Forster and Larchey-Wendling [FL19] as a slight variation of the definition given by Minsky [Min67, Table 11.1-1]. In particular, the conditional jump on decrement is on a strictly positive counter value (and not on zero). Intuitively, conditional jumps on zero (as in Minsky's original definition) severely restrict meaningful control flow when only two counters are available. In fact, in the setting with only two counters the halting problem for the original definition is *decidable* [Dud22a, Remark 1], and for universality requires additional control flow instructions.

Despite its remarkable simplicity, the halting problem for two-counter machines (Problem 2.3) is undecidable (Corollary 2.5).

Problem 2.3 (Two-counter Machine Halting)**.** Given a two-counter machine $\mathcal{M}$ and two natural numbers $a, b \in \mathbb{N}$, does the configuration $(0, (a, b))$ terminate in $\mathcal{M}$?

Lemma 2.4. The Turing machine halting problem many-one reduces to the two-counter machine halting problem (Problem 2.3).

Proof Sketch. Minsky describes the simulation of Turing machines by machines with four counters [Min67, Section 11.2] working on Gödel encodings. Then, machines with four counters are simulated by two-counter machines [Min67, Theorem 14.1-1]. □

Corollary 2.5. Two-counter machine halting (Problem 2.3) is undecidable.

Remark 2.6. Minsky's original argument is constructive and, despite some confusion (Remark 2.2), is sufficient for our technical result. However, to avoid ambiguity with respect to the instruction set used, for mechanization we rely on existing work by Forster and Larchey-Wendling [FL19]. This approach emerged as part of the effort to mechanize Hilbert's tenth problem [LF22]. Specifically, it many-one reduces Turing machine halting via the Post correspondence problem [Pos46, FHS18] to n-counter machine halting, and ultimately to two-counter machine halting.

Remark 2.7. Besides Minsky's original definition, there are several different universal instruction sets for two-counter machines [Kor96, Section 2]. The particular choice in the present work is motivated by the following two facts. First, the instruction set is sufficiently restricted[6]. This allows for shorter proofs based on reductions *from* properties of the particular machine model. Second, the existing mechanization for the undecidability of the corresponding halting problem is well-maintained, and is readily available in the Coq Library of Undecidability Proofs.

Commonly, two-counter machines are easily simulated by other machine models, and therefore serve a key role in undecidability proofs for machine immortality and boundedness problems [Hoo66, KO08]. However, they have one drawback with respect to boundedness properties. Specifically, there is no natural increasing measure on configurations along the step relation, as it allows for cycles (Example 2.8).

Example 2.8. Consider the two-counter machine $\mathcal{M} = [\mathtt{inc}_0, \mathtt{dec}_0\, 0]$. For any counter values $a, b \in \mathbb{N}$ the configuration $(0, (a, b))$ does not terminate in $\mathcal{M}$ because of the cycle

$$(0, (a, b)) \longrightarrow_{\mathcal{M}} (1, (a + 1, b)) \longrightarrow_{\mathcal{M}} (0, (a, b)) \longrightarrow_{\mathcal{M}} \dots$$

Still, the configuration $(0, (a, b))$ is *bounded*, because from it only a finite number of distinct configurations is reachable. In fact, from any configuration at most 2 distinct configurations are reachable in $\mathcal{M}$.

One could eliminate cycles by introducing a third counter, which increases at every step. This establishes an equivalence between mortality and corresponding boundedness properties. While inducing a natural increasing measure (value of the third counter), it incurs additional bookkeeping. A simulation of such a three-counter machine by an acyclic two-counter machine is possible [KO08], however, it again obscures the underlying measure.

We address this drawback of two-counter machines with respect to boundedness properties in the following Section 3, building upon Minsky's notion of machines with *one* counter.

[6] One can combine the $\mathtt{inc}_0$ and the $\mathtt{inc}_1$ instructions to one $\mathtt{inc}_{\{0,1\}}$ instruction. However, this is insignificant in practice.

3. One-counter Machines

As Minsky originally observed [Min67, Section 14.2], with multiplication and division by constants the halting problem is undecidable for *one-counter machines*. Specifically, increase and decrease operations for two values a and b can be simulated by multiplication and division respectively by 2 and 3 respectively for one value $2^a 3^b$.

In this section, we further develop Minsky's construction (similarly to [Woj99]) of universal machines with one counter in pursuit of two goals. First, machine runs should be easy to simulate in the stack machine model (Remark 3.11) used in Section 4. Second, we require a measure on machine configurations that increases along the step relation, directly connecting non-termination and unboundedness (Lemma 3.5). Both goals are motivated by the effort necessary to mechanize the overarching result[7].

A program instruction of a one-counter machine (Definition 3.1), besides modifying the program index, conditionally multiplies the current counter value with either $\frac{2}{1}$, $\frac{3}{2}$, $\frac{4}{3}$, or $\frac{5}{4}$. Notably, such a multiplication by $\frac{d+1}{d}$ for $d \in \{1, 2, 3, 4\}$ is both easy to simulate uniformly, and strictly increases a positive counter value.

Definition 3.1 (One-counter Machine ($\mathcal{P}$))**.** A *one-counter machine* $\mathcal{P}$ is a list of *instructions* of shape (j, d), where $j \in \mathbb{N}$ is a *program index* and $d \in \{1, 2, 3, 4\}$ is a *counter modifier*.

A *configuration* of $\mathcal{P}$ is a pair (i, c), where $i \in \mathbb{N}$ is the current *program index* and $c \in \mathbb{N}$ such that $c > 0$ is the current *counter value*.

The *step relation* of $\mathcal{P}$ on configurations, written $(\longrightarrow_{\mathcal{P}})$, is given by

- if $|\mathcal{P}| \leq i$, then $(i, c) \longrightarrow_{\mathcal{P}} (i, c)$ and we say (i, c) is *halting*;
- if (j, d) is at index i in $\mathcal{P}$ and d divides c, then $(i, c) \longrightarrow_{\mathcal{P}} (j, c \cdot \frac{d+1}{d})$;
- if (j, d) is at index i in $\mathcal{P}$ and d does not divide c, then $(i, c) \longrightarrow_{\mathcal{P}} (i + 1, c)$.

The *n-fold step relation* is denoted $\longrightarrow^n_{\mathcal{P}}$. The *reachability relation* of $\mathcal{P}$ on configurations, written $(\longrightarrow^*_{\mathcal{P}})$, is the reflexive, transitive closure of $(\longrightarrow_{\mathcal{P}})$.

A configuration (i, c) *terminates* in $\mathcal{P}$, if we have $(i, c) \longrightarrow^*_{\mathcal{P}} (i', c')$ for some halting configuration (i', c').

Intuitively, the fractions $\frac{2}{1}, \frac{3}{2}, \frac{4}{3}, \frac{5}{4}$ serve the following goals. The fractions $\frac{2}{1}$ and $\frac{3}{2}$ multiply by 2 and 3 respectively. The fractions $\frac{3}{2}$ and $\frac{4}{3}$ divide by 2 and 3 respectively. The fraction $\frac{5}{4}$ removes the superfluous factor 4 introduced by the fraction $\frac{4}{3}$.

Example 3.2. Consider the one-counter machine $\mathcal{P} = [(1, 1), (0, 2)]$. The configuration $(0, 1)$ does not terminate in $\mathcal{P}$ because of the infinite configuration chain

$$(0,1) \longrightarrow_{\mathcal{P}} (1, 1 \cdot \frac{2}{1}) \longrightarrow_{\mathcal{P}} (0, 2 \cdot \frac{3}{2}) \longrightarrow_{\mathcal{P}} (1, 3 \cdot \frac{2}{1}) \longrightarrow_{\mathcal{P}} (0, 6 \cdot \frac{3}{2}) \longrightarrow_{\mathcal{P}} (1, 9 \cdot \frac{2}{1}) \longrightarrow_{\mathcal{P}} \cdots$$

Naturally, the counter value strictly increases in the above configuration chain.

Remark 3.3. One-counter machines can be understood as a variant of Conway's FRACTRAN language [Con87] with a relaxed program index transition rule, and restricted to the instruction set $(\frac{2}{1}, \frac{3}{2}, \frac{4}{3}, \frac{5}{4})$.

The step relation for one-counter machines is total (Lemma 3.4(1)), deterministic (Lemma 3.4(2)), and forms increasing chains (Lemma 3.4(4) and Lemma 3.4(5)) up to a halting configuration, which is necessarily a trivial configuration loop (Lemma 3.4(3)).

[7]The author firmly believes that proof surveyability does improve with a simpler mechanization.

Lemma 3.4 (One-counter Machine Step Relation Properties)**.**

(1) **Totality:**
For all configurations (i, c) there is a configuration (i', c') such that $(i, c) \longrightarrow_{\mathcal{P}} (i', c')$.
(2) **Determinism:**
If $(i, c) \longrightarrow_{\mathcal{P}} (i', c')$ and $(i, c) \longrightarrow_{\mathcal{P}} (i'', c'')$, then $(i', c') = (i'', c'')$.
(3) **Halting Property:**
A configuration (i, c) is halting iff $(i, c) \longrightarrow_{\mathcal{P}} (i, c)$.
(4) **Increasing Measure:**
If $(i, c) \longrightarrow_{\mathcal{P}} (i', c')$ and (i, c) is not halting, then $|\mathcal{P}| \cdot c + i < |\mathcal{P}| \cdot c' + i'$.
(5) **Monotone Counter:**
(a) If $(i, c) \longrightarrow_{\mathcal{P}} (i', c')$, then $c \leq c'$.
(b) If $(i, c) \longrightarrow_{\mathcal{P}}^{|\mathcal{P}|+1} (i', c')$ and (i', c') is not halting, then $c < c'$.

Proof. Routine case analysis. □

The above Lemma 3.4(4) gives a strictly increasing measure $|\mathcal{P}| \cdot c + i$ on non-halting configurations (i, c) with respect to the step relation $(\longrightarrow_{\mathcal{P}})$. Therefore, any configuration cycle is trivial, i.e. the corresponding configuration is halting. Additionally, by Lemma 3.4(5), the counter value is guaranteed to increase after $|\mathcal{P}| + 1$ steps, unless a halting configuration is reached. This results in a characterization of termination via boundedness of reachable counter values (Lemma 3.5).

Lemma 3.5. Let $\mathcal{P}$ be a one-counter machine. A configuration (i, c) terminates in $\mathcal{P}$ iff there is a $k \in \mathbb{N}$ such that for all configurations (i', c') with $(i, c) \longrightarrow_{\mathcal{P}}^{*} (i', c')$ we have $c' < k$.

Proof. If from the configuration (i, c) the machine $\mathcal{P}$ halts after n steps, then $k = 1 + c \cdot 2^n$ bounds the reachable from (i, c) counter values. Conversely, if k bounds the reachable from the configuration (i, c) counter values, then after at most $k \cdot (|\mathcal{P}| + 1)$ steps a halting configuration is reached by Lemma 3.4(5). □

The halting problem for one-counter machines (Problem 3.6) starting from the fixed[8] configuration $(0, 1)$ is undecidable by reduction from the halting problem for two-counter machines (Lemma 3.7).

Problem 3.6 (One-counter Machine 1-Halting)**.** Given a one-counter machine $\mathcal{P}$, does the configuration $(0, 1)$ terminate in $\mathcal{P}$?

Lemma 3.7. Two-counter machine halting (Problem 2.3) many-one reduces to one-counter machine 1-halting (Problem 3.6).

Proof. Let $\mathcal{M}$ be a two-counter machine and let $a_0, b_0 \in \mathbb{N}$ be two starting counter values.

We represent a pair (a, b) of counter values of $\mathcal{M}$ by the family of counter values $2^a 3^b 5^m$ where $m \in \mathbb{N}$. The purpose of the factor 5^m is to increase whenever either the factor 2^a or the factor 3^b decreases. We simulate instructions of $\mathcal{M}$ on two counters (a, b) by instructions of $\mathcal{P}$ on one counter $c = 2^a 3^b 5^m$ as follows.

Increase a**:** $2^a 3^b 5^m \cdot \frac{2}{1} = 2^{a+1} 3^b 5^m$;
Increase b**:** $2^a 3^b 5^m \cdot \frac{2}{1} \cdot \frac{3}{2} = 2^a 3^{b+1} 5^m$;
Decrease a**:** $2^{a+1} 3^b 5^m \cdot \frac{3}{2} \cdot \frac{4}{3} \cdot \frac{5}{4} = 2^a 3^b 5^{m+1}$;
Decrease b**:** $2^a 3^{b+1} 5^m \cdot \frac{4}{3} \cdot \frac{5}{4} = 2^a 3^b 5^{m+1}$.

[8]We fix a starting configuration in order to obtain a simpler recursive simulation in the proof of Lemma 4.16.

Simulated decrease instructions are followed by a simulation of an unconditional jump instruction via $2^a3^b5^m \cdot \frac{2}{1} \cdot \frac{2}{1} \cdot \frac{5}{4} = 2^a3^b5^{m+1}$, such that on failed decrease the simulation can continue with an appropriate program index. Initialization of counter values (a_0, b_0) is simulated via $2^03^05^0 \cdot (\frac{2}{1})^{a_0+b_0} \cdot (\frac{3}{2})^{b_0} = 2^{a_0}3^{b_0}5^0$.

Overall, the configuration $(0, (a_0, b_0))$ terminates in the two-counter machine $\mathcal{M}$ iff the configuration $(0, 1)$ terminates in the one-counter machine $\mathcal{P}$. □

Corollary 3.8. One-counter machine 1-halting (Problem 3.6) is undecidable.

The following Example 3.9 illustrates the construction in the proof of Lemma 3.7, simulating a looping two-counter machine from Example 2.8.

Example 3.9. Consider the two-counter machine $\mathcal{M} = [\mathtt{inc}_0, \mathtt{dec}_0\, 0]$ from Example 2.8 with the initial counter values $(a_0, b_0) = (1, 1)$. Following the proof of Lemma 3.7, we construct the one-counter machine

$$\mathcal{P} = [(1,1), (2,1), (3,2), (4,1), (5,2), (6,3), (3,4)]$$

Starting from the configuration $(0, 1) = (0, 2^03^05^0)$ a run of $\mathcal{P}$ starts with the initialization

$$(0, 2^03^05^0) \xrightarrow{(1,1)}_{\mathcal{P}} (1, 2^13^05^0) \xrightarrow{(2,1)}_{\mathcal{P}} (2, 2^23^05^0) \xrightarrow{(3,2)}_{\mathcal{P}} (3, 2^13^15^0) = (3, 2^{a_0}3^{b_0}5^0)$$

Next, on iteration of the infinite loop of $\mathcal{M}$ is simulated, returning to the program index 3

$$(3, 2^13^15^0) \xrightarrow{(4,1)}_{\mathcal{P}} (4, 2^23^15^0) \xrightarrow{(5,2)}_{\mathcal{P}} (5, 2^13^25^0) \xrightarrow{(6,3)}_{\mathcal{P}} (6, 2^33^15^0) \xrightarrow{(3,4)}_{\mathcal{P}} (3, 2^13^15^1)$$

While the factors 2^1 and 3^1 remain unchanged after the above simulated iteration, the factor 5^0 strictly increases to 5^1, which is systematic.

As a result, the configuration $(0, 1) = (0, 2^03^05^0)$ does not terminate in $\mathcal{P}$, simulating non-termination of the configuration $(0, (a_0, b_0))$ in $\mathcal{M}$ as follows

$$(0, 2^03^05^0) \longrightarrow^3_{\mathcal{P}} (3, 2^13^15^0) \longrightarrow^4_{\mathcal{P}} (3, 2^13^15^1) \longrightarrow^4_{\mathcal{P}} (3, 2^13^15^2) \longrightarrow^4_{\mathcal{P}} (3, 2^13^15^3) \longrightarrow^4_{\mathcal{P}} \dots$$

Indeed, there is no upper bound on the counter value, in congruence with Lemma 3.5.

Remark 3.10. Another universal collection of fractions is $\frac{6}{1}, \frac{1}{2}, \frac{1}{3}$ [Min67, Exercise 14.2-2]. However, this collection allows for non-trivial loops and does not induce an increasing measure along the step relation.

Remark 3.11. It is possible to use the fractions $\frac{6}{1}$, $\frac{5}{2}$, and $\frac{5}{3}$ to establish an increasing measure. In comparison, the deliberate choice of counter multiplication by $\frac{d+1}{d}$ for a counter modifier $d \in \{1, 2, 3, 4\}$ has several benefits. First, instructions are of uniform shape. Therefore, simulation of and reasoning about such instructions requires less case analysis, also impacting the underlying mechanization. Second, for a counter value $c = k \cdot d$ multiplication by $\frac{d+1}{d}$ results in $c \cdot \frac{d+1}{d} = c + k$, which is easy to simulate in the two-stack machine model.

4. Two-stack Machines

In this section, our goal is the simulation of one-counter machines in a *stack machine* model of computation without unbounded *symbol search*. Specifically, given a one-counter machine $\mathcal{P}$, we construct a two-stack machine $\mathcal{S}$ such that the configuration $(0, 1)$ terminates in $\mathcal{P}$ iff there is a uniform bound on the number of configurations reachable in $\mathcal{S}$ from any configuration.

The main difficulty, similar to the undecidability proof for Turing machine immortality [Hoo66], is to simulate symbol search (traverse data, searching for a particular symbol) using a uniformly bounded machine. Most problematic are unsuccessful searches that may traverse an arbitrary, i.e. not uniformly bounded, amount of data. The key idea [Hoo66, Part IV] (see also [KO08, Jea12]) is to implement unbounded symbol search by nested bounded symbol search.

In the present work, we supplement a high level explanation of the construction (cf. proof of Lemma 4.16) with a comprehensive case analysis as a mechanized proof in the Coq proof assistant. This approach, arguably worth striving for in general, has three advantages over existing work. First, the proof idea is not cluttered with mundane technical details, while the mechanized proof is highly precise. Second, a mechanized proof leaves little doubt regarding proof correctness and is open to scrutiny, as there is nothing left to imagination. Third, the Coq proof assistant tracks any non-constructive assumptions which may hide beneath technical details.

Let us specify the two-stack machine (Definition 4.1) model of computation, which we use to simulate one-counter machines. An instruction of such a machine may modify the current machine state, pop from, and push onto two stacks of binary symbols. We denote the empty word by ϵ, and the concatenation of words A and B by AB.

Definition 4.1 (Two-stack Machine ($\mathcal{S}$))**.** Let $\mathbb{S}$ be a countably infinite set. A *two-stack machine* $\mathcal{S}$ is a list of *instructions* of shape $A\shortmid p\shortmid B \to A'\shortmid q\shortmid B'$, where $A, B, A', B' \in \{0,1\}^*$ are binary *words* and $p, q \in \mathbb{S}$ are *states*.

A *configuration* of $\mathcal{S}$ is of shape $A\shortmid p\shortmid B$ where $p \in \mathbb{S}$ is the *current state*, $A \in \{0,1\}^*$ is the (reversed) content of the *left stack* and $B \in \{0,1\}^*$ is the content of the *right stack*.

The *step relation* of $\mathcal{S}$ on configurations, written $(\longrightarrow_{\mathcal{S}})$, is given by

- if $(A\shortmid p\shortmid B \to A'\shortmid q\shortmid B') \in \mathcal{S}$, then for $C, D \in \{0,1\}^*$ we have $CA\shortmid p\shortmid BD \longrightarrow_{\mathcal{S}} CA'\shortmid q\shortmid B'D$.

The *reachability relation* of $\mathcal{S}$ on configurations, written $(\longrightarrow_{\mathcal{S}}^*)$, is the reflexive, transitive closure of $(\longrightarrow_{\mathcal{S}})$.

Remark 4.2. A two-stack machine can be understood as a restricted semi-Thue system on the alphabet $\{0,1\} \cup \mathbb{S}$ in which each word contains exactly one symbol from $\mathbb{S}$. Such rewriting systems are employed in the setting of synchronous distributivity [AEL$^+$12].

Remark 4.3. To accommodate for arbitrary large machines, the state space $\mathbb{S}$ is not finite. However, the *effective* state space of any two-stack machine $\mathcal{S}$ is bounded by the finitely many states occurring in the instructions of $\mathcal{S}$ (a list of finite length).

The key undecidable property of two-stack machines, used in [Dud20], is whether the number of distinct, reachable configurations from any configuration is *uniformly bounded* (Definition 4.4).

Definition 4.4 (Uniformly Bounded)**.** A two-stack machine $\mathcal{S}$ is *uniformly bounded* if there exists an $n \in \mathbb{N}$ such that for any configuration $A\shortmid p\shortmid B$ we have

$$|\{A'\shortmid p'\shortmid B' \mid A\shortmid p\shortmid B \longrightarrow_{\mathcal{S}}^* A'\shortmid p'\shortmid B'\}| \leq n$$

Notably, uniform boundedness and *uniform termination* [MS96] (is every configuration chain finite?) are orthogonal notions, illustrated by the following Examples 4.5–4.8.

Example 4.5. Consider the empty two-stack machine $\mathcal{S} = []$. It is uniformly bounded by $n = 1$ and does uniformly terminate as it admits only singleton configuration chains.

Example 4.6. Consider the two-stack machine $\mathcal{S} = [0\mid p\mid\epsilon \to \epsilon\mid p\mid 1]$. From the configuration $0^m\mid p\mid\epsilon$, where $m \in \mathbb{N}$, reachable in $\mathcal{S}$ configurations are exactly $0^{m-i}\mid p\mid 1^i$ for $i = 0, \ldots, m$. Therefore, there is no *uniform* bound on the number of reachable configurations. However, the length of every configuration chain in $\mathcal{S}$ is finite (bounded by one plus the length of the left stack). Overall, $\mathcal{S}$ does uniformly terminate but is not uniformly bounded.

Example 4.7. Consider the two-stack machine $\mathcal{S} = [(0\mid p\mid\epsilon \to \epsilon\mid q\mid 1), (\epsilon\mid q\mid 1 \to 0\mid p\mid\epsilon)]$. The number of distinct configurations reachable in $\mathcal{S}$ from any configuration is uniformly bounded by $n = 2$. However, $\mathcal{S}$ admits an infinite configuration chain

$$0\mid p\mid\epsilon \longrightarrow_{\mathcal{S}} \epsilon\mid q\mid 1 \longrightarrow_{\mathcal{S}} 0\mid p\mid\epsilon \longrightarrow_{\mathcal{S}} \epsilon\mid q\mid 1 \longrightarrow_{\mathcal{S}} \ldots$$

In summary, $\mathcal{S}$ is uniformly bounded, but does not uniformly terminate.

Example 4.8. Consider the two-stack machine $\mathcal{S} = [0\mid p\mid\epsilon \to \epsilon\mid p\mid 1, \epsilon\mid p\mid 1 \to 0\mid p\mid\epsilon]$. Similarly to Example 4.6, $\mathcal{S}$ is not uniformly bounded. Additionally, similarly to Example 4.7, it admits an infinite configuration chain

$$0\mid p\mid\epsilon \longrightarrow_{\mathcal{S}} \epsilon\mid p\mid 1 \longrightarrow_{\mathcal{S}} 0\mid p\mid\epsilon \longrightarrow_{\mathcal{S}} \epsilon\mid p\mid 1 \longrightarrow_{\mathcal{S}} \ldots$$

In summary, $\mathcal{S}$ is neither uniformly bounded nor does uniformly terminate.

In literature [Hoo66, KO08], counter machine termination is simulated using uniformly bounded Turing machines directly rather than by two-stack machines. This is reasonable when omitting technical details regarding the exact Turing machine construction. However, for verification in full detail, Turing machines are quite unwieldy, compared to two-stack machines. Unfortunately, we cannot rely on existing mechanized Turing machine programming techniques [FKW20], as they establish only functional properties, and are incapable to establish uniform boundedness.

4.1. Deterministic, Length-preserving Two-stack Machines.

There are several properties of two-stack machines that are of importance in our construction in order to reuse existing work [Dud20].

For *length-preserving* two-stack machines (Definition 4.9) the sum of lengths of the two stacks is invariant wrt. reachability. For each configuration, length-preservation bounds (albeit, not uniformly) the number of distinct, reachable configurations. Therefore, reachability is decidable (in polynomial space) for length-preserving two-stack machines.

Definition 4.9 (Length-preserving)**.** A two-stack machine $\mathcal{S}$ is *length-preserving* if for all instructions $(A\mid p\mid B \to A'\mid q\mid B') \in \mathcal{S}$ we have $0 < |A| + |B| = |A'| + |B'|$.

Definition 4.10 (Deterministic)**.** A two-stack machine $\mathcal{S}$ is *deterministic* if for all configurations $A\mid p\mid B$, $A'\mid p'\mid B'$, and $A''\mid p''\mid B''$ such that $A\mid p\mid B \longrightarrow_{\mathcal{S}} A'\mid p'\mid B'$ and $A\mid p\mid B \longrightarrow_{\mathcal{S}} A''\mid p''\mid B''$ we have $A'\mid p'\mid B' = A''\mid p''\mid B''$.

Example 4.11. Two-stack machines in Examples 4.5–4.7 are deterministic and length-preserving. The two-stack machine $\mathcal{S} = [0\mid p\mid\epsilon \to \epsilon\mid p\mid 1, \epsilon\mid p\mid 1 \to 0\mid p\mid\epsilon]$ from Example 4.8 is length-preserving, but not deterministic because of $0\mid p\mid 1 \longrightarrow_{\mathcal{S}} \epsilon\mid p\mid 11$ and $0\mid p\mid 1 \longrightarrow_{\mathcal{S}} 00\mid p\mid\epsilon$.

Remark 4.12. Deterministic, length-preserving two-stack machines can be understood as a generalization of intercell Turing machines [KTU93b, Section 3], for which the transition rule has a bounded read/write/move radius around the current head position. Tape content on the left (resp. right) of the current head position is exactly the content of the left (resp. right) stack.

The key undecidable problem, that in our argument assumes the role of Turing machine immortality of previous approaches [KTU93b, Dud20], is uniform boundedness of deterministic, length-preserving two-stack machines (Problem 4.13). A central insight of the present work is that using this problem as an intermediate step we neither require Turing reductions, König's lemma (cf. [KTU93b]), nor the fan theorem (cf. [Dud20]). Additionally, a mechanization for both the reduction from counter machine halting to this problem and the reduction from this problem to semi-unification is of manageable size.

Problem 4.13 (Deterministic, Length-preserving Two-stack Machine Uniform Boundedness)**.** Given a deterministic, length-preserving two-stack machine $\mathcal{S}$, is $\mathcal{S}$ uniformly bounded?

The original undecidability proof of semi-unification contains a hint [KTU93b, Proof of Corollary 6] that Turing machine immortality may be avoided. Accordingly, Lemma 4.16 captures the decisive step, which avoids Turing machine immortality.

Remark 4.14 (Naive Simulation)**.** Given a one-counter machine $\mathcal{P}$, we can construct a length-preserving two-stack machine $\mathcal{S}$ such that $(0,1)$ terminates in $\mathcal{P}$ iff configurations $A1\imath 0\imath 01B$ for $A, B \in \{0,1\}^*$ are uniformly bounded in $\mathcal{S}$.

A $\mathcal{P}$-configuration (i,c) can be represented by the $\mathcal{S}$-configuration $1\imath i\imath 0^c 10^m$, where $m \in \mathbb{N}$, such that 0^m is long enough to simulate a terminating run. In particular, the counter value c is unary encoded as 0^c1 with 1 as separator. A $\mathcal{P}$-instruction (j,d) at index i is simulated as follows. The $\mathcal{S}$-instruction $\epsilon\imath i_?\imath 0^d \longrightarrow_{\mathcal{S}} 0^d\imath i_?\imath\epsilon$ tests for divisibility by d, moving consecutive blocks of d zeroes from the right stack to the left stack. The divisibility test either fails or succeeds.

Failure $(i,c) \longrightarrow_{\mathcal{P}} (i+1,c)$**:** The $\mathcal{S}$-instruction $\epsilon\imath i_?\imath 0^k1 \longrightarrow_{\mathcal{S}} \epsilon\imath i_\#\imath 0^k1$ can be applied, where k is the remainder such that $0 < k < d$. In this case, we move the zeroes back to the right stack, and via the $\mathcal{S}$-instruction $1\imath i_\#\imath\epsilon \longrightarrow_{\mathcal{S}} 1\imath i+1\imath\epsilon$ we reach the $\mathcal{S}$-configuration $1\imath i+1\imath 0^c10^m$, representing the next $\mathcal{P}$-configuration $(i+1,c)$.

Success $(i,c) \longrightarrow_{\mathcal{P}} (j, c\cdot\frac{d+1}{d})$**:** The $\mathcal{S}$-instruction $\epsilon\imath i_?\imath 1 \longrightarrow_{\mathcal{S}} \epsilon\imath i_!\imath 1$ can be applied. In this case, multiplication by $\frac{d+1}{d}$ is simulated as follows. For each block of consecutive d zeroes on the left stack shift the separator 1 on the right stack one position to the right (Remark 3.11). Finally, via the $\mathcal{S}$-instruction $1\imath i_!\imath\epsilon \longrightarrow_{\mathcal{S}} 1\imath j\imath\epsilon$, we arrive at the $\mathcal{S}$-configuration $1\imath j\imath 0^{\frac{c(d+1)}{d}}10^{m-\frac{c}{d}}$, representing the next $\mathcal{P}$-configuration $(j, c\cdot\frac{d+1}{d})$.

We obtain $(i,c) \longrightarrow^*_{\mathcal{P}} (i',c')$ iff $A1\imath i\imath 0^c10^{c'-c}B \longrightarrow^*_{\mathcal{S}} A1\imath i'\imath 0^{c'}1B$ for all words $A, B \in \{0,1\}^*$. Therefore, the configuration $(0,1)$ terminates in $\mathcal{P}$ iff configurations $A1\imath 0\imath 01B$ are uniformly bounded in $\mathcal{S}$ (the uniform bound is derived from the halting counter value).

Unfortunately, the naive construction in the above Remark 4.14 in general does not describe a uniformly bounded two-stack machine. Most importantly, the naive construction assumes and preserves the "safe" format $A1\imath i\imath 0^c1B$ of machine configurations (i,c), where c is less or equal to the halting counter value. Arbitrary configurations do not need to follow this format. At fault is a symbol search (for the symbol 1 on the right stack) that needs to traverse an arbitrary amount of data, illustrated by the following Example 4.15.

Example 4.15. Consider a one-counter machine $\mathcal{P}$ containing the instruction (j,d) at index i. The naive construction of a two-stack machine $\mathcal{S}$ from Remark 4.14 contains the $\mathcal{S}$-instruction $\epsilon\imath i_?\imath 0^d \longrightarrow_{\mathcal{S}} 0^d\imath i_?\imath\epsilon$. For any bound $n \in \mathbb{N}$ we have that $|\{A\imath p\imath B \mid 1\imath i_?\imath 0^{n\cdot d}1 \longrightarrow^*_{\mathcal{S}} A\imath p\imath B\}| > n$. Therefore, $\mathcal{S}$ is not uniformly bounded.

The ingenious idea by Hooper [Hoo66, Part IV] is to use *nested simulation* for symbol search. In the present scenario, to search for the symbol 1 on the right stack, start a new simulation from the $\mathcal{P}$-configuration $(0, 1)$ inside the space of consecutive zeroes on the right stack. Nested simulation achieves three goals. First, it transitions into a "safe" configuration format $A1_{\mid i \mid}0^c1B$, where c is less or equal to the halting counter value. Second, by inspecting symbols in the immediate neighborhood of the separator 1 on the right stack, it checks whether the original search for the symbol 1 succeeds in appropriately limited space. Third, in case the space limit for symbol search is exceeded, it simulates a terminating run of the given machine.

Similarly to [KO08, Theorem 7], we adapt Hooper's nested simulation for symbol search in the following Lemma 4.16.

Lemma 4.16. One-counter machine 1-halting (Problem 3.6) many-one reduces to deterministic, length-preserving two-stack machine uniform boundedness (Problem 4.13).

Proof. Let $\mathcal{P}$ be a one-counter machine. We follow the naive construction in Remark 4.14 with the following difference. In order to search for the symbol 1 on the right stack in some configuration $A_{\mid p \mid}0^{k+3}B$, transition into the configuration $AC1_{\mid 0 \mid}01B$, i.e. reset the program index p to 0 and retain the binary encoding of p in C of fixed length k.

Assume that the $\mathcal{P}$-configuration $(0, 1)$ terminates in $\mathcal{P}$ and let c be the counter value of the halting configuration. For a nested simulation from the configuration $AC1_{\mid 0 \mid}01B$ in search of symbol 1 in B, there are three cases.

Case $B = 0^m1D$, where $m < c - 1$ and $D \in \{0, 1\}^*$: The number m of zeroes on the right stack is too small to accommodate for c. Eventually, the nested simulation is unable to simulate counter increase. This is detected by inspecting symbols in the immediate neighborhood of the separator 1 on the right stack. In this situation, the initial search for the symbol 1 in B succeeds, the previous level configuration is restored, and control is returned to the previous level. The content of D is not accessed, and symbol search succeeds in bounded space (the bound is derived from $m < c - 1$).

Case $B = 0^m$, where $m < c - 1$: The size of the right stack is too small to accommodate for c. Eventually, the nested simulation is unable to shift the separator 1 to the right (in fact, apply any instruction), and halts. In this situation, the initial search for the symbol 1 fails in bounded space (the bound is derived from $m < c - 1$).

Since the original configuration $A_{\mid p \mid}0^{k+3}B$ is not in the "safe" format (missing separator 1 on the right stack), this case handles ill-formed configurations in bounded space, and the search is immaterial.

Case $B = 0^{c-1}D$, where $D \in \{0, 1\}^*$: There are enough consecutive zeroes in B to simulate a terminating run of $\mathcal{P}$ from the initial configuration $(0, 1)$. The content of D is not accessed, and the simulation terminates in bounded space (the bound is derived from $c - 1$).

Since the original configuration $A_{\mid p \mid}0^{k+3}B$ is not in the "safe" format (too many consecutive zeroes on the right stack), this case handles ill-formed configurations in bounded space, and the search is immaterial.

Each case may require further symbol search, and therefore introduce further nested simulation. Notably, consecutive nested simulation is performed inside the space of at most c consecutive zeroes (followed by the separator 1 in the "safe" format). Therefore, the nesting depth is at most c. In each case (using Lemma 3.4) a bound on the number of configurations for the nested simulation can be derived from c. Therefore, $\mathcal{S}$ is uniformly bounded.

For the converse, assume that the $\mathcal{P}$-configuration $(0, 1)$ does not terminate in $\mathcal{P}$. By Lemma 3.5, there is no bound on reachable counter values. Therefore, there is no uniform bound on the number of configurations reachable from configurations $1\mid 0\mid 010^m$, where $m \in \mathbb{N}$ is arbitrary large. □

Corollary 4.17. Deterministic, length-preserving two-stack machine uniform boundedness (Problem 4.13) is undecidable.

Remark 4.18. At first glance, recursive nested simulation in the proof of Lemma 4.16 seems superfluous. After all, a nested simulation establishes a "safe" configuration format with bounded (naive) symbol search. However, it is not possible to remember reliably in the current configuration (e.g. in the current state) whether a nested simulation is running. An ill-formed configuration may be locally indistinguishable from a configuration of a nested simulation. Therefore, any (recursive) symbol search is performed using nested simulation.

Remark 4.19. The exact analysis of the nested simulation in the proof of Lemma 4.16 requires a tremendous inductive proof with many corner cases. Arguably, it is unreasonable for a human without mechanical assistance to write it down in full detail (cf. Hooper's remark in Section 1). Additionally, it would require a comparable amount of effort for others to verify such a massive construction. This is why, in order to guarantee its correctness, a mechanized proof of Lemma 4.16 is adequate (Section 6) to establish the result.

4.2. **Confluent, Simple Two-stack Machines.** To further simplify the construction, we consider confluent (instead of deterministic), simple (Definition 4.20, cf. [Dud20, Definition 16]) two-stack machines. Uniform boundedness for such two-stack machines (Problem 4.26) is well-suited for reduction to a fragment of semi-unification (cf. Section 5.1).

Definition 4.20 (Simple)**.** A two-stack machine $\mathcal{S}$ is *simple* if for all instructions $(A\mid p\mid B \to A'\mid q\mid B') \in \mathcal{S}$ we have $1 = |A| + |B| = |A'| + |B'| = |A| + |A'| = |B| + |B'|$.

Remark 4.21. A deterministic, simple two-stack machine is just another way to present a deterministic Turing machine. The left and right stacks contain the respective tape content to the left and to the right from the current head. Reading and writing at the head while moving the head position is easily presented as simple instructions (cf. [Dud20, Remark 19]).

Remark 4.22. Turing machine immortality is reducible to uniform boundedness of deterministic, simple two-stack machines by a bounded Turing reduction [Dud20, Theorem 2]. However, the argument uses the fan theorem, therefore, it is crucial for the present argument to not rely on this particular reduction.

Definition 4.23 (Confluent)**.** A two-stack machine $\mathcal{S}$ is *confluent* if for all configurations $A\mid p\mid B$, $A'\mid p'\mid B'$, and $A''\mid p''\mid B''$ such that $A\mid p\mid B \longrightarrow^*_{\mathcal{S}} A'\mid p'\mid B'$ and $A\mid p\mid B \longrightarrow^*_{\mathcal{S}} A''\mid p''\mid B''$ there exists a configuration $C\mid q\mid D$ such that $A'\mid p'\mid B' \longrightarrow^*_{\mathcal{S}} C\mid q\mid D$ and $A''\mid p''\mid B'' \longrightarrow^*_{\mathcal{S}} C\mid q\mid D$.

Any deterministic two-stack machine is confluent (Lemma 4.24), but not necessarily vice versa (Example 4.25).

Lemma 4.24. If a two-stack machine $\mathcal{S}$ is deterministic, then $\mathcal{S}$ is confluent.

Proof. For a deterministic two-stack machine $\mathcal{S}$ we have that $A\mid p\mid B \longrightarrow^*_{\mathcal{S}} A'\mid p'\mid B'$ and $A\mid p\mid B \longrightarrow^*_{\mathcal{S}} A''\mid p''\mid B''$ implies that $A'\mid p'\mid B' \longrightarrow^*_{\mathcal{S}} A''\mid p''\mid B''$ or $A''\mid p''\mid B'' \longrightarrow^*_{\mathcal{S}} A'\mid p'\mid B'$. □

Example 4.25. The two-stack machine $\mathcal{S} = [0\vert p\vert\epsilon \to \epsilon\vert p\vert 1, \epsilon\vert p\vert 1 \to 0\vert p\vert\epsilon]$ from Example 4.8 is non-deterministic, as observed in Example 4.11. However, $\mathcal{S}$ is confluent. The instruction $0\vert p\vert\epsilon \to \epsilon\vert p\vert 1$ can be "undone" by the instruction $\epsilon\vert p\vert 1 \to 0\vert p\vert\epsilon$ and vice versa. Therefore, configuration chains can be reversed to join branching computation. This technique is used in the proof of Lemma 4.27.

Compared to deterministic machines, confluent machines are quite practical. For example, a confluent machine may (without additional bookkeeping) "try out" different configuration chains before choosing the preferable one (cf. proof of Lemma 4.27).

Problem 4.26 (Confluent, Simple Two-stack Machine Uniform Boundedness)**.** Given a confluent, simple two-stack machine $\mathcal{S}$, is $\mathcal{S}$ uniformly bounded?

By Lemma 4.24, the above Problem 4.26 subsumes deterministic, simple two-stack machine uniform boundedness [Dud20, Problem 26]. This, in combination with the following Lemma 4.27, allows for adaptation of previous work[9] in Section 5.1.

Lemma 4.27. Deterministic, length-preserving two-stack machine uniform boundedness (Problem 4.13) many-one reduces to confluent, simple two-stack machine uniform boundedness (Problem 4.26).

Proof. Our objective is to shorten length-preserving instructions, while maintaining confluence. This is routine, storing local stack information in additional fresh states. For example, an instruction $00\vert p\vert\epsilon \longrightarrow 11\vert q\vert\epsilon$ can be replaced by the simple instructions $0\vert p\vert\epsilon \longrightarrow \epsilon\vert p_1\vert 0$, $0\vert p_1\vert\epsilon \longrightarrow \epsilon\vert p_2\vert 0$, $\epsilon\vert p_2\vert 0 \longrightarrow 1\vert p_3\vert\epsilon$, and $\epsilon\vert p_3\vert 0 \longrightarrow 1\vert q\vert\epsilon$, where p_1, p_2, p_3 are fresh states. This results in the configuration chain

$$00\vert p\vert\epsilon \longrightarrow 0\vert p_1\vert 0 \longrightarrow \epsilon\vert p_2\vert 00 \longrightarrow 1\vert p_3\vert 0 \longrightarrow 11\vert q\vert\epsilon$$

which simulates the instruction $00\vert p\vert\epsilon \longrightarrow 11\vert q\vert\epsilon$.

Notably, it is difficult to maintain determinism for failed look-ahead, when we want to preserve elegance of the above simulation. However, in order to maintain confluence it suffices to add reverse instructions from fresh states, that is $\epsilon\vert p_1\vert 0 \longrightarrow 0\vert p\vert\epsilon$, $\epsilon\vert p_2\vert 0 \longrightarrow 0\vert p_1\vert\epsilon$, and $1\vert p_3\vert\epsilon \longrightarrow \epsilon\vert p_2\vert 0$. Therefore, any failed attempt to read local stack information can be reversed (cf. Example 4.25) and computation is confluent. □

Corollary 4.28. Confluent, simple two-stack machine uniform boundedness (Problem 4.26) is undecidable.

Finally, let us justify the detour from one-counter machines via deterministic, length-preserving two-stack machines to confluent, simple two-stack machines. By far the most complicated part of the overall reduction is the simulation of a terminating machine by a uniformly bounded machine (cf. Lemma 4.16). For this reduction, it is beneficial to have a restricted source machine model and an expressive target machine model. Of course, a *direct* simulation of terminating one-counter machines by uniformly bounded, confluent, simple two-stack machines is possible. However, the restricted nature of simple instructions would induce an unwieldy two-stack machine construction, rendering the mechanization more laborious.

[9]It is possible to carry out the construction in the original, deterministic scenario without adaptation. However, this is technically more challenging and provides no tangible benefit.

5. Semi-unification

Semi-unification (Problem 5.3) can be understood as a combination of first-order unification (cf. substitution φ) and first-order matching (cf. substitutions ψ). For the undecidability of semi-unification [KTU93a, Theorem 12], it suffices to restrict the syntax of the underlying terms (Definition 5.1) to variables together with a single binary constructor ($\to$).

In this section, we recapitulate necessary definitions and properties of semi-unification from existing work [KTU93b, Dud20], in order to complete a constructive many-one reduction from Turing machine halting to semi-unification (Theorem 5.23).

Definition 5.1 (Terms ($\mathbb{T}$))**.** Let α, β, γ range over a countably infinite set $\mathbb{V}$ of *variables*. The set of *terms* $\mathbb{T}$, ranged over by σ, τ, is given by the following grammar

$$\sigma, \tau \in \mathbb{T} ::= \alpha \mid \sigma \to \tau$$

Definition 5.2 (Substitution (φ), (ψ))**.** A *substitution* $\varphi : \mathbb{V} \to \mathbb{T}$ assigns terms to variables, and is tacitly lifted to terms.

Problem 5.3 (Semi-unification [KTU93b, SUP], [Dud20, Problem 3])**.**
Given a set $\mathcal{I} = \{\sigma_1 \leq \tau_1, \ldots, \sigma_n \leq \tau_n\}$ of *inequalities*, is there a substitution φ such that for each inequality $(\sigma \leq \tau) \in \mathcal{I}$ there exists a substitution $\psi : \mathbb{V} \to \mathbb{T}$ such that $\psi(\varphi(\sigma)) = \varphi(\tau)$?

Remark 5.4. As given by Definition 5.2, the set of substitutions is not countable. However, in any semi-unification instance $\mathcal{I}$ the number of inequalities (consisting of first-order terms) is finite. Therefore, restricting substitutions to be finite maps (from the relevant variables) does not change the expressive power of semi-unification. As a result, semi-unification is recursively enumerable.

The following Example 5.5 and Example 5.6 illustrate a positive and negative instances of semi-unification.

Example 5.5. Consider $\mathcal{I} = \{\alpha \leq \alpha \to \alpha,\ \alpha \leq \alpha \to \alpha \to \alpha\}$. The semi-unification instance $\mathcal{I}$ is solved by the substitution φ such that $\varphi(\alpha) = \alpha$.

- For the inequality $\alpha \leq \alpha \to \alpha$ there exists a substitution ψ_1 such that $\psi_1(\alpha) = \alpha \to \alpha$. We have $\psi_1(\varphi(\alpha)) = \varphi(\alpha \to \alpha)$.
- For the inequality $\alpha \leq \alpha \to \alpha \to \alpha$ there exists a (different from ψ_1) substitution ψ_2 such that $\psi_2(\alpha) = \alpha \to \alpha \to \alpha$. We have $\psi_2(\varphi(\alpha)) = \varphi(\alpha \to \alpha \to \alpha)$.

Example 5.6. Consider $\mathcal{I} = \{\alpha \to \alpha \leq \alpha\}$. The semi-unification instance $\mathcal{I}$ has no solution. Assume that there exist substitutions φ and ψ such that $\psi(\varphi(\alpha \to \alpha)) = \varphi(\alpha)$. Therefore, the size of the syntax tree of $\varphi(\alpha)$ is twice the size of the syntax tree $\psi(\varphi(\alpha))$ which is not possible for (non-empty, finite) terms.

The following Example 5.7 compares type inference for *parametric polymorphism* [Mil78] based on unification, and for *recursive polymorphism* [Myc84] based on semi-unification.

Example 5.7. Consider the following functional program `f`, where `not` is Boolean negation

```
f b x = if b then x else f (f (not b) (not b)) x
```

Basically, `f b x` reduces to `x`, regardless of the value of `b`. However, both `x` and `not b` are used as second argument for some recursive call of `f`.

For languages with *parametric polymorphism*, recursive calls are considered *monomorphic*. Therefore, type inference for `f` unifies the type of `x` and the type of `not b`. This results in the inferred type $\texttt{f} : \texttt{bool} \to \texttt{bool} \to \texttt{bool}$.

For languages with *recursive polymorphism* the type of each individual recursive call can be instantiated separately. The problematic recursive call `f (not b) (not b)` is associated with the inequality $\sigma \leq \tau$, where $\sigma = \texttt{bool} \to \alpha \to \alpha$ is the global type scheme and $\tau = \texttt{bool} \to \texttt{bool} \to \texttt{bool}$ is the local type scheme. Any substitution φ such that $\varphi(\alpha) = \beta$ for some $\beta \in \mathbb{V}$ solves the inequality $\sigma \leq \tau$ setting $\psi(\beta) = \texttt{bool}$. That is, we have $\psi(\varphi(\sigma)) = \varphi(\tau)$. This results in the inferred type $\texttt{f} : \texttt{bool} \to \alpha \to \alpha$.

Unfortunately, semi-unification does not admit a decision procedure based on an occurs-check, which is a common approach to both first-order unification and first-order matching [BSN+01]. However, it is challenging to construct an unsolvable example of manageable size, for which the occurs-check is not triggered.

Originally [KTU93b, Theorem 12], semi-unification is proven undecidable by Turing reduction from Turing machine immortality [Hoo66]. As intermediate problems, the argument relies on symmetric intercell Turing machine boundedness, path equation derivability, and termination of a custom-tailored redex contraction procedure. Additionally, the argument uses König's lemma and it is not obvious whether it can be presented constructively.

A modern approach [Dud20, Theorem 4] simplifies the original argument. It still relies on a Turing reduction from Turing machine immortality, but uses only deterministic, simple two-stack machine uniform boundedness to show undecidability of a fragment of semi-unification. Additionally, it relies on the fan theorem, which is strictly weaker than König's lemma and is valid in Brouwer's intuitionism. The argument is partially mechanized in Coq.

In the remainder of this section we briefly recapitulate and reuse the modern approach [Dud20] in the more general case of confluent, simple two-stack machines. This allows us to avoid Turing machine immortality, Turing reductions, and the fan theorem in the overall argument.

5.1. Simple Semi-unification.

In this section, we recapitulate the intermediate problem of *simple semi-unification* (Problem 5.10) [Dud20, Problem 15], which connects stack machine computation and semi-unification. Intuitively, term variables represent machine states, simple constraints (Definition 5.8) represent local stack transformations, and the model relation (Definition 5.9) captures machine reachability via substitutions.

Definition 5.8 (Simple Constraint [Dud20, Definition 6])**.** A *simple constraint* has the shape $a \mid \alpha \mid \epsilon \doteq \epsilon \mid \beta \mid b$, where $a, b \in \{0, 1\}$ are symbols and $\alpha, \beta \in \mathbb{V}$ are variables.

Definition 5.9 (Model Relation [Dud20, Definition 9])**.** A substitution triple $(\varphi, \psi_0, \psi_1)$ *models* a simple constraint $a \mid \alpha \mid \epsilon \doteq \epsilon \mid \beta \mid b$, written $(\varphi, \psi_0, \psi_1) \models a \mid \alpha \mid \epsilon \doteq \epsilon \mid \beta \mid b$, if one of the following conditions holds

- $b = 0$ and $\psi_a(\varphi(\alpha)) \to \tau = \varphi(\beta)$ for some term $\tau \in \mathbb{T}$;
- $b = 1$ and $\sigma \to \psi_a(\varphi(\alpha)) = \varphi(\beta)$ for some term $\sigma \in \mathbb{T}$.

The intuition behind a simple constraint $a \mid \alpha \mid \epsilon \doteq \epsilon \mid \beta \mid b$ is as follows. The symbol $a \in \{0, 1\}$ specifies the associated substitution ψ_a. This corresponds to considering two inequalities [KTU93a, Remark after Theorem 12]. The symbol $b \in \{0, 1\}$ specifies whether we are interested in the left of the right subterm of $\varphi(\beta)$. This allows for arbitrary deep exploration of term structure.

Problem 5.10 (Simple Semi-unification [Dud20, Problem 15])**.** Given a finite set $\mathcal{C}$ of simple constraints, do there exist substitutions $\varphi, \psi_0, \psi_1 : \mathbb{V} \to \mathbb{T}$ such that for each simple constraint $(a \shortmid \alpha \shortmid \epsilon \doteq \epsilon \shortmid \beta \shortmid b) \in \mathcal{C}$ we have $(\varphi, \psi_0, \psi_1) \models a \shortmid \alpha \shortmid \epsilon \doteq \epsilon \shortmid \beta \shortmid b$?

The following Examples 5.11–5.12 illustrate a positive and a negative instance of simple semi-unification.

Example 5.11. Consider the simple constraints $\mathcal{C} = \{0 \shortmid \alpha \shortmid \epsilon \doteq \epsilon \shortmid \beta \shortmid 1\}$. A possible substitution triple $(\varphi, \psi_0, \psi_1)$ which models $0 \shortmid \alpha \shortmid \epsilon \doteq \epsilon \shortmid \beta \shortmid 1$ is such that $\varphi(\alpha) = \alpha$, $\varphi(\beta) = \beta_1 \to \beta_2$, and $\psi_0(\alpha) = \beta_2$. Indeed, we have $\beta_1 \to \psi_0(\varphi(\alpha)) = \beta_1 \to \beta_2 = \varphi(\beta)$. In fact, $\mathcal{C}$ is related to the uniformly bounded two-stack machine $\mathcal{S} = [(0 \shortmid p \shortmid \epsilon \to \epsilon \shortmid q \shortmid 1), (\epsilon \shortmid q \shortmid 1 \to 0 \shortmid p \shortmid \epsilon)]$ from Example 4.7.

Example 5.12. Consider the simple constraints $\mathcal{C} = \{0 \shortmid \alpha \shortmid \epsilon \doteq \epsilon \shortmid \alpha \shortmid 1\}$. There is no substitution triple $(\varphi, \psi_0, \psi_1)$ which models $0 \shortmid \alpha \shortmid \epsilon \doteq \epsilon \shortmid \alpha \shortmid 1$. For some σ such a substitution triple would satisfy $\sigma \to \psi_0(\varphi(\alpha)) = \varphi(\alpha)$. This is not possible because the syntax tree of $\sigma \to \psi_0(\varphi(\alpha))$ is strictly larger than the syntax tree of $\varphi(\alpha)$. In fact, $\mathcal{C}$ is related to the two-stack machine $\mathcal{S} = [0 \shortmid p \shortmid \epsilon \to \epsilon \shortmid p \shortmid 1]$, which is not uniformly bounded, from Example 4.6.

The pivotal result in previous work [Dud20, Section 4] is a many-one reduction from deterministic, simple two-stack machine uniform boundedness to simple semi-unification. We recapitulate this result in the confluent case (Lemma 5.13).

Lemma 5.13. Confluent, simple two-stack machine uniform boundedness (Problem 4.26) many-one reduces to simple semi-unification (Problem 5.10).

Proof. We tacitly inject machine states into variables, i.e. $\mathbb{S} \subseteq \mathbb{V}$. Given a confluent, simple two-stack machine $\mathcal{S}$, we construct the set of simple constraints $\mathcal{C}$ [Dud20, Definition 40]:

$$\mathcal{C} = \{a \shortmid p \shortmid \epsilon \doteq \epsilon \shortmid q \shortmid b \mid (a \shortmid p \shortmid \epsilon \to \epsilon \shortmid q \shortmid b) \in \mathcal{S} \text{ or } (\epsilon \shortmid q \shortmid b \to a \shortmid p \shortmid \epsilon) \in \mathcal{S}\}$$

[Dud20, Lemma 45]**:** If $\mathcal{S}$ is uniformly bounded, then there exists a substitution triple $(\varphi, \psi_0, \psi_1)$ which models each simple constraint in $\mathcal{C}$.

[Dud20, Lemma 48]**:** If a substitution triple $(\varphi, \psi_0, \psi_1)$ models each constraint in $\mathcal{C}$, then the maximal depth of the syntax trees in the range of φ induces a uniform bound on the number of configurations reachable from any configuration in $\mathcal{S}$. □

Corollary 5.14. Simple semi-unification (Problem 5.10) is undecidable.

Remark 5.15. Although the original construction [Dud20, Lemma 45 and Lemma 48] is based on determinism, only confluence is used in the actual proofs [Dud20, Lemma 30 and Lemma 39]. While tedious to verify by hand, the available mechanization allows for simple replacement of determinism by confluence. Meanwhile, the proof assistant guarantees correctness of (or indicates problems with) any related details. This highlights the effectiveness of proof assistants to accommodate for changes in a complex argument, reevaluating overall correctness.

5.2. Right/Left-Uniform, Two-inequality Semi-unification.

In this section, we consider a restriction of semi-unification to only two inequalities with identical right-hand sides (Problem 5.16).

Problem 5.16 (Right-uniform, Two-inequality Semi-unification)**.** Given two inequalities $\sigma_0 \leq \tau$ and $\sigma_1 \leq \tau$ with identical right-hand sides, do there exist substitutions φ, ψ_0, ψ_1 such that $\psi_0(\varphi(\sigma_0)) = \varphi(\tau)$ and $\psi_1(\varphi(\sigma_1)) = \varphi(\tau)$?

Remark 5.17. Simply put, the above Problem 5.16 can be stated as follows: Given three terms σ_0, σ_1, τ, are there substitutions φ, ψ_0, ψ_1 such that $\psi_0(\varphi(\sigma_0)) = \varphi(\tau) = \psi_1(\varphi(\sigma_1))$?

We adjust the existing reduction from simple semi-unification to (non right-uniform) two-inequality semi-unification [Dud20, Theorem 1] to produce right-uniform inequalities.

Lemma 5.18. Simple semi-unification (Problem 5.10) many-one reduces to right-uniform, two-inequality semi-unification (Problem 5.16).

Proof. Given constraints $\mathcal{C} = \{a_i \shortmid \alpha_i \shortmid \epsilon \doteq \epsilon \shortmid \beta_i \shortmid b_i \mid i = 1, \dots, n\}$, define $\tau = \beta_1 \to \cdots \to \beta_n$. Let γ_i be fresh variables for $i = 1, \dots, n$ and define $\sigma^j = \sigma^j_1 \to \cdots \to \sigma^j_n$ for $j \in \{0, 1\}$ where

$$\sigma^j_i = \alpha_i \to \gamma_i \text{ if } a_i = j \text{ and } b_i = 0 \qquad \sigma^j_i = \gamma_i \to \alpha_i \text{ if } a_i = j \text{ and } b_i = 1 \qquad \sigma^j_i = \gamma_i \text{ else}$$

We show that $\mathcal{C}$ is solvable iff the right-uniform inequalities $\sigma^0 \le \tau$ and $\sigma^1 \le \tau$ are solvable.

- Assume that the substitution triple $(\varphi, \psi_0, \psi_1)$ models each simple constraint in $\mathcal{C}$. W.l.o.g. $\varphi(\gamma_i) = \psi_0(\gamma_i) = \psi_1(\gamma_i) = \gamma_i$ for $i = 1, \dots, n$. By routine case analysis, we may adjust $\psi_0(\gamma_i)$ and $\psi_1(\gamma_i)$ for $i = 1, \dots, n$ to obtain substitutions ψ'_0 and ψ'_1 such that $\psi'_0(\varphi(\sigma^0)) = \varphi(\tau) = \psi'_1(\varphi(\sigma^1))$.
- Any solution φ, ψ_0, ψ_1 of $\sigma^0 \le \tau$ and $\sigma^1 \le \tau$ also models each constraint in $\mathcal{C}$. □

Corollary 5.19. Right-uniform, two-inequality semi-unification (Problem 5.16) is undecidable.

Akin to the right-uniform (identical right-hand sides) restriction, it is straightforward to formulate *left-uniform* (identical left-hand sides), two-inequality semi-unification [Dud21, Problem 34], and prove its undecidability. This is used as a starting point in a recent, alternative undecidability proof of System F typability [Dud21, Lemma 40].

Problem 5.20 (Left-uniform, Two-inequality Semi-unification)**.** Given two inequalities $\sigma \le \tau_0$ and $\sigma \le \tau_1$ with identical left-hand sides, do there exist substitutions φ, ψ_0, ψ_1 such that $\psi_0(\varphi(\sigma)) = \varphi(\tau_0)$ and $\psi_1(\varphi(\sigma)) = \varphi(\tau_1)$?

Lemma 5.21. Right-uniform two-inequality semi-unification (Problem 5.16) many-one reduces to left-uniform, two-inequality semi-unification (Problem 5.20).

Proof. Given two right-uniform inequalities $\sigma_0 \le \tau$ and $\sigma_1 \le \tau$, let α_0, α_1 be fresh variables. Construct the terms $\sigma' := \sigma_0 \to \sigma_1$, $\tau'_0 := \tau \to \alpha_1$, and $\tau'_1 := \alpha_0 \to \tau$. We show that $\sigma_0 \le \tau$ and $\sigma_1 \le \tau$ are solvable iff the left-uniform inequalities $\sigma' \le \tau'_0$ and $\sigma' \le \tau'_1$ are solvable.

- Assume $\psi_0(\varphi(\sigma_0)) = \varphi(\tau)$ and $\psi_1(\varphi(\sigma_1)) = \varphi(\tau)$. Construct the substitution φ' such that $\varphi'(\alpha_0) = \psi_1(\varphi(\sigma_0))$, $\varphi'(\alpha_1) = \psi_0(\varphi(\sigma_1))$, and otherwise $\varphi'(\alpha) = \varphi(\alpha)$.
  We have $\psi_0(\varphi'(\sigma')) = \varphi(\tau) \to \varphi'(\alpha_1) = \varphi'(\tau'_0)$ and $\psi_1(\varphi'(\sigma')) = \varphi'(\alpha_0) \to \varphi(\tau) = \varphi'(\tau'_1)$. Therefore, φ', ψ_0, ψ_1 solve the left-uniform inequalities $\sigma' \le \tau'_0$ and $\sigma' \le \tau'_1$.
- Assume $\psi_0(\varphi(\underbrace{\sigma'}_{\sigma_0 \to \sigma_1})) = \varphi(\underbrace{\tau'_0}_{\tau \to \alpha_1})$ and $\psi_1(\varphi(\underbrace{\sigma'}_{\sigma_0 \to \sigma_1})) = \varphi(\underbrace{\tau'_1}_{\alpha_0 \to \tau})$. Immediately, we have $\psi_0(\varphi(\sigma_0)) = \varphi(\tau) = \psi_1(\varphi(\sigma_1))$. □

Corollary 5.22. Left-uniform, two-inequality semi-unification (Problem 5.20) is undecidable.

5.3. **Main Result.** Finally, we compose the previously described reductions into a comprehensive, constructive many-one reduction from Turing machine halting to semi-unification (Theorem 5.23). This constitutes the main result of the present work.

Theorem 5.23. Turing machine halting constructively many-one reduces to semi-unification (Problem 5.3).

Proof. By composition of Lemmas 2.4, 3.7, 4.16, 4.27, 5.13, and 5.18. Constructivity of the argument is witnessed by an axiom-free mechanization (Section 6) using the Coq proof assistant. □

Since semi-unification is recursively enumerable (Remark 5.4), it is RE-complete under many-one reductions (in the sense of [Rog87, Chapter 7.2]).

Corollary 5.24. Semi-unification (Problem 5.3) is RE-complete under many-one reductions.

6. Mechanization

This section provides an overview over the constructive mechanization, using the Coq proof assistant [Coqa], of the many-one reduction from Turing machine halting to semi-unification. The mechanization relies on, and is integrated into the growing Coq Library of Undecidability Proofs [FLWD+20]. The reduction is axiom-free and spans approximately 20000 lines of code, of which 3500 is contributed by the present work.

At the core of the library is the following mechanized notion of many-one reducibility[10]

```
Definition reduction {X Y} (f: X -> Y) (P: X -> Prop) (Q: Y -> Prop) :=
  forall x, P x <-> Q (f x).

Definition reduces {X Y} (P: X -> Prop) (Q: Y -> Prop) :=
  exists f: X -> Y, reduction f P Q.

Notation "P ⪯ Q" := (reduces P Q).
```

In the above, a predicate `P` over the domain `X` many-one reduces to a predicate `Q` over the domain `Y`, denoted `P ⪯ Q`, if there exists a function `f: X -> Y` such that for all `x` in the domain `X` we have `P x` iff `Q (f x)`. Implemented in axiom-free Coq any such function `f: X -> Y` is computable, a necessity oftentimes handled with less rigor in traditional (non-mechanized) proofs. Additionally, an axiom-free proof of `P x <-> Q (f x)` cannot rely on principles such as functional extensionality, the fan theorem, or the law of excluded middle.

Overall, the statement `Theorem reduction : HaltTM 1 ⪯ SemiU`[11] faithfully mechanizes the overall formal goal of a constructive many-one reduction from Turing machine halting to semi-unification (Theorem 5.23). Correctness of the argument is witnessed by verification in axiom-free Coq, for which constructivity is certified using the `Print Assumptions` command [Coqb]. In fact, the particular many-one reduction function could be extracted from the proof of `HaltTM 1 ⪯ SemiU` as a λ-term (in the call-by-value λ-calculus model of computation) using existing techniques [FK19].

[10] `theories/Synthetic/Definitions.v` in the github repository of [FLWD+20]

[11] `theories/SemiUnification/Reductions/HaltTM_1_to_SemiU.v`

In detail, key contributions of the present work are consolidated as part of the following conjunction of many-one reductions[12].

```
Theorem HaltTM_1_chain_SemiU :
  HaltTM 1 ⪯ iPCPb /\
  iPCPb ⪯ Halt_BSM /\
  Halt_BSM ⪯ MM2_HALTING /\
  MM2_HALTING ⪯ CM1_HALT /\
  CM1_HALT ⪯ SMNdl_UB /\
  SMNdl_UB ⪯ CSSM_UB /\
  CSSM_UB ⪯ SSemiU /\
  SSemiU ⪯ RU2SemiU /\
  RU2SemiU ⪯ SemiU.
```

The individual problems in the above chain of many-one reductions are as follows.

- `HaltTM 1` is one-tape Turing machine halting, and native to the library as an initial undecidable problem, building upon prior work [FKW20, AR15] in computability theory;
- `iPCPb` is indexed, binary Post correspondence problem, mechanized in [FHS18];
- `Halt_BSM` is binary stack machine halting, mechanized in [FL19];
- `MM2_HALTING` is two-counter machine halting (Problem 2.3), mechanized in [FL19];
- `CM1_HALT` is one-counter machine 1-halting (Problem 3.6);
- `SMNdl_UB` is uniform boundedness of deterministic, length-preserving two-stack machines (Problem 4.13);
- `CSSM_UB` is uniform boundedness of confluent, simple stack machines (Problem 4.26);
- `SSemiU` is simple semi-unification (Problem 5.10), mechanized in [Dud20];
- `RU2SemiU` is right-uniform, two-inequality semi-unification (Problem 5.16);
- `SemiU` is semi-unification (Problem 5.3), mechanized in [Dud20].

In the remainder of this section we sketch the contributed mechanizations of `CM1_HALT`, `SMNdl_UB`, and `CSSM_UB` together with corresponding many-one reductions.

6.1. **One-counter Machines.** A one-counter machine `Cm1`[13] is mechanized as a list of instructions `Instruction`, which are pairs `State * nat` of a program index and a counter modifier. According to Definition 3.1, `State` is set to `nat`. Since a counter modifier is strictly positive, an instruction (j, d) (Definition 3.1) is mechanized as the pair `(j, d-1)`.

```
Definition State := nat.

Record Config := mkConfig { state: State; value: nat }.

Definition Instruction := State * nat.

Definition Cm1 := list Instruction.
```

The function `step` mechanizes the step function on configurations, which are records with two fields: `state` (program index) and `value` (counter value). The function `nth_error` retrieves the current instruction `(p, n)`. On failure, `step` enters a trivial loop, mechanized as `halting`.

1-halting (Problem 3.6) is mechanized as `CM1_HALT`, that is the existence of a number of steps `n` after which the iterated step function reaches a halting configuration starting

[12] `theories/SemiUnification/Reductions/HaltTM_1_chain_SemiU.v`

[13] `theories/CounterMachines/CM1.v`

from the configuration `{| state := 0; value := 1 |}`. The counter modifier of one-counter machines `M` in `CM1_HALT` is less than 4, mechanized as `Forall (fun '(_, n) => n < 4) M`.

```
Definition step (M: Cm1) (x: Config) : Config :=
  match (value x), (nth_error M (state x)) with
  | 0, _ => x (* halting configuration *)
  | _, None => x (* halting configuration *)
  | _, Some (p, n) =>
      match modulo (value x) (n+1) with
      | 0 => {| state := p; value := ((value x) * (n+2)) / (n+1) |}
      | _ => {| state := 1 + state x; value := value x |}
      end
  end.

Definition halting (M: Cm1) (x: Config) := step M x = x.

Definition CM1_HALT : {M: Cm1 | Forall (fun '(_, n) => n < 4) M} -> Prop :=
  fun '(exist _ M _) =>
    exists n, halting M (Nat.iter n (step M) {| state := 0; value := 1 |}).
```

The proof of `MM2_HALTING` $\preceq$ `CM1_HALT`[14] (Lemma 3.7) is by straightforward simulation and spans in total approximately 400 lines of code.

6.2. **Deterministic, Length-preserving Two-stack Machines.** A two-stack machine `SMN`[15] is mechanized as a list of instructions `Instruction` containing transition information. The step relation `step` on configurations `Config` holds for instructions `((r, s, x), (r', s', y))` such that `x` is the current state, `y` is the next state, `r`, `s` are the prefixes of the current respective left and right stacks, and `r'`, `s'` are the prefixes of the next respective left and right stacks. The state space `State` is set to `nat` (we could have chosen any effectively enumerable type with decidable equality). Configuration reachability `reachable` is mechanized as the reflexive, transitive closure of `step`.

```
Definition State := nat.

Definition Symbol := bool.

Definition Stack := list Symbol.

Definition Config := Stack * Stack * State.

Definition Instruction := Config * Config.

Definition SMN := list Instruction.

Inductive step (M: SMN) : Config -> Config -> Prop :=
  | transition (v w r s r' s': Stack) (x y: State) :
    In ((r, s, x), (r', s', y)) M ->
    step M (r ++ v, s ++ w, x) (r' ++ v, s' ++ w, y).

Definition reachable (M: SMN) : Config -> Config -> Prop :=
  clos_refl_trans Config (step M).
```

[14] theories/CounterMachines/Reductions/MM2_HALTING_to_CM1_HALT.v

[15] theories/StackMachines/SMN.v

A deterministic, length-preserving two-stack machine is mechanized as a dependent pair `{M : SMN | deterministic M /\ length_preserving M}` of a two-stack machine `M` and a pair of (irrelevant) proofs showing that `M` is deterministic and length-preserving. Uniform boundedness (Problem 4.13) is mechanized as `SMNdl_UB`, that is the existence of maximal length `n` of exhaustive lists `L` of reachable configurations `Y` from any given configuration `X`.

```
Definition bounded (M: SMN) (n: nat) : Prop :=
  forall (X: Config), exists (L: list Config),
    (forall (Y: Config), reachable M X Y -> In Y L) /\ length L <= n.

Definition length_preserving (M: SMN) : Prop :=
  forall s t X s' t' Y, In ((s, t, X), (s', t', Y)) M ->
    length (s ++ t) = length (s' ++ t') /\ 1 <= length (s ++ t).

Definition SMNdl_UB :
  {M : SMN | deterministic M /\ length_preserving M} -> Prop :=
    fun '(exist _ M _) => exists (n: nat), bounded M n.
```

The proof of `CM1_HALT` $\preceq$ `SMNdl_UB`[16] (Lemma 4.16) relies on a variant of Hooper's argument [Hoo66]. The particular mechanization details span approximately 2300 lines of code, two thirds of which mechanize Hooper's argument[17]. As a side note, the mechanization does *not* construct a "mirror" machine [Hoo66, Part IV] for symbol search on the left stack (symmetric to symbol search on the right stack). Instead, machine instructions may swap stacks. This simplifies the construction and does not change uniform boundedness. The particular machine `M` consists of instructions simulating divisibility tests `index_ops`, counter increase `increase_ops`, stack traversal `goto_ops`, and nested simulation initialization `bound_ops`.

```
Definition M : SMX :=
  locked index_ops ++ locked increase_ops ++
  locked goto_ops ++ locked bound_ops.
```

Since the proof structure for uniform bound verification is mostly by induction, extensive case analysis, and basic arithmetic, the mechanization benefits greatly from proof automation, i.e. Coq's `lia`, `nia`, and `eauto` tactics [Coqb]. For example, the proof automation tactic `eauto with M` is used to solve machine specification correctness obligations. It uses type-directed backwards search and relies on a hint database of instruction specifications for the constructed machine `M`.

To the best of the author's knowledge, the provided mechanization is the first that implements (a variant of) Hooper's argument.

6.3. **Confluent Two-stack Machines.** A confluent, simple two-stack machine `cssm`[18] is mechanized as a dependent pair `{M : ssm | confluent M}` of a simple two-stack machine `M` and an (irrelevant) proof that `M` is confluent.

```
Definition confluent (M: ssm) := forall (X Y1 Y2: config),
  reachable M X Y1 -> reachable M X Y2 ->
    exists (Z: config), reachable M Y1 Z /\ reachable M Y2 Z.

Definition cssm := {M: ssm | confluent M}.
```

[16] `theories/StackMachines/Reductions/CM1_HALT_to_SMNdl_UB.v`

[17] `theories/StackMachines/Reductions/CM1_HALT_to_SMNdl_UB/CM1_to_SMX.v`

[18] `theories/StackMachines/SSM.v`

Uniform boundedness (Problem 4.26) is mechanized as `CSSM_UB`, that is the existence of maximal length `n` of exhaustive lists `L` of reachable configurations `Y` from any given configuration `X`.

```
Definition bounded (M: ssm) (n: nat) :=
  forall (X: config), exists (L: list config),
    (forall (Y: config), reachable M X Y -> In Y L) /\ length L <= n.

Definition CSSM_UB (M: cssm) := exists (n: nat), bounded (proj1_sig M) n.
```

The proof of `SMNdl_UB` $\preceq$ `CSSM_UB`[19] (Lemma 5.13) is by revertible (confluent), local lookahead and spans approximately 800 lines of code.

The proof of `CSSM_UB` $\preceq$ `SSemiU`[20] (Lemma 5.13) is an almost verbatim copy of the corresponding proof of `DSSM_UB` $\preceq$ `SSemiU` from prior work [Dud20, Section 5], in which determinism is replaced by confluence. Most importantly, the key function ζ [Dud20, Definition 41] is used to directly construct simple semi-unification solutions.

7. Conclusion

This work gives a constructive many-one reduction from Turing machine halting to semi-unification (Theorem 5.23). It improves upon existing work [KTU93b, Dud20] regarding the following aspects.

First, previous approaches use Turing reductions to establish undecidability. Therefore, such arguments are unable to establish RE-completeness under many-one reductions of semi-unification, shown in the present work (Corollary 5.24).

Second, previous work relies on the undecidability of Turing machine immortality, which is not recursively enumerable, and obscures the overall picture. In the present work, we avoid Turing machine immortality by adapting Hooper's ingenious construction [Hoo66] (also adapted in [KO08]) to uniform boundedness (Lemma 4.16).

Third, correctness of the reduction function is proven constructively (in the sense of axiom-free Coq), whereas previous work uses the principle of excluded middle, König's lemma [KTU93b], or the fan theorem [Dud20]. As a result, anti-classical theories, such as synthetic computability theory [Bau05], may accommodate the presented results.

Fourth, computability of the many-one reduction function from Turing machines to semi-unification instances is established rigorously by its mechanization in the Coq proof assistant. Traditionally, this aspect is treated less formally.

Fifth, the reduction is mechanized and contributed to the Coq Library of Undecidability Proofs [FLWD$^+$20], building upon existing infrastructure. Arguably, a comprehensive mechanization is the *only* feasible approach to verify a reduction from Turing machine halting to semi-unification with high confidence in full detail. The provided mechanization integrates existing work [Dud20] into the library, and contributes a first of its kind mechanized variant of Hooper's construction for uniformly bounded symbol search.

While this document provides a high-level overview over the overall argument, surveyability (both local and global in the sense of [Bas06]) is established mechanically. Local surveyability is supported by the modular nature of the Coq Library of Undecidability Proofs. That is, the mechanization of each reduction step can be understood and verified

[19] `theories/StackMachines/Reductions/SMNdl_UB_to_CSSM_UB.v`

[20] `theories/SemiUnification/Reductions/CSSM_UB_to_SSemiU.v`

independently. Global surveyability is supported by `Theorem HaltTM_1_chain_SemiU` and the statement `HaltTM 1` $\preceq$ `SemiU` (cf. Section 6). That is, the individually mechanized reduction steps do compose transitively.

The provided mechanization shows the maturity of the Coq proof assistant for machine-assisted verification of technically challenging proofs. Neither Hooper's exact immortality construction [Hoo66] nor the exact semi-unification construction by Kfoury, Tiuryn, and Urzyczyn [KTU93b] was mechanized. Rather, the overall structure of the proof was revised to be mechanization-friendly. For example, the simplicity and uniformity of one-counter machines as an intermediate model of computation serves exactly this purpose.

Already, building upon the present work, there is a novel mechanization showing the undecidability of System F typability and type checking [Dud21]. In addition, we envision further mechanized results. For one, the undecidability of synchronous distributivity [AEL+12] relies on uniform boundedness of semi-Thue systems that can be described as the presented (and mechanized) two-stack machines. Further, since the underlying construction is already implemented, it is reasonable to mechanize a many-one reduction from Turing machine halting to Turing machine immortality. This would pave the way for further mechanized results. For example, the undecidability of the finite variant property [BGLN13, Section 7] as well as several tiling problems [Kar07] rely on (variants of) Turing machine immortality.

References

[AEL+12] Siva Anantharaman, Serdar Erbatur, Christopher Lynch, Paliath Narendran, and Michaël Rusinowitch. Unification modulo synchronous distributivity. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, volume 7364 of *Lecture Notes in Computer Science*, pages 14–29. Springer, 2012. doi:10.1007/978-3-642-31365-3_4.

[AR15] Andrea Asperti and Wilmer Ricciotti. A formalization of multi-tape Turing machines. *Theor. Comput. Sci.*, 603:23–42, 2015. doi:10.1016/j.tcs.2015.07.013.

[Bas06] O. Bradley Bassler. The surveyability of mathematical proof: A historical perspective. *Synth.*, 148(1):99–133, 2006. doi:10.1007/s11229-004-6221-7.

[Bau05] Andrej Bauer. First steps in synthetic computability theory. In Martín Hötzel Escardó, Achim Jung, and Michael W. Mislove, editors, *Proceedings of the 21st Annual Conference on Mathematical Foundations of Programming Semantics, MFPS 2005, Birmingham, UK, May 18-21, 2005*, volume 155 of *Electronic Notes in Theoretical Computer Science*, pages 5–31. Elsevier, 2005. doi:10.1016/j.entcs.2005.11.049.

[BGLN13] Christopher Bouchard, Kimberly A. Gero, Christopher Lynch, and Paliath Narendran. On forward closure and the finite variant property. In Pascal Fontaine, Christophe Ringeissen, and Renate A. Schmidt, editors, *Frontiers of Combining Systems - 9th International Symposium, FroCoS 2013, Nancy, France, September 18-20, 2013. Proceedings*, volume 8152 of *Lecture Notes in Computer Science*, pages 327–342. Springer, 2013. doi:10.1007/978-3-642-40885-4_23.

[BSN+01] Franz Baader, Wayne Snyder, Paliath Narendran, Manfred Schmidt-Schauß, and Klaus U. Schulz. Unification theory. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 445–532. Elsevier and MIT Press, 2001. doi:10.1016/b978-044450813-3/50010-2.

[Con87] John H. Conway. Fractran: A simple universal programming language for arithmetic. In *Open Problems in Communication and Computation*, pages 4–26. Springer, 1987.

[Coqa] The Coq Proof Assistant. https://coq.inria.fr/. Accessed: 2023-09-11.

[Coqb] The Coq Proof Assistant Reference Manual. https://coq.inria.fr/distrib/current/refman/. Accessed: 2023-09-11.

[DM82] Luís Damas and Robin Milner. Principal type-schemes for functional programs. In Richard A. DeMillo, editor, *Conference Record of the Ninth Annual ACM Symposium on Principles of*

Programming Languages, Albuquerque, New Mexico, USA, January 1982, pages 207–212. ACM Press, 1982. doi:10.1145/582153.582176.

[Dud20] Andrej Dudenhefner. Undecidability of semi-unification on a napkin. In Zena M. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, June 29-July 6, 2020, Paris, France (Virtual Conference)*, volume 167 of *LIPIcs*, pages 9:1–9:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.FSCD.2020.9.

[Dud21] Andrej Dudenhefner. The undecidability of system F typability and type checking for reductionists. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021*, pages 1–10. IEEE, 2021. doi:10.1109/LICS52264.2021.9470520.

[Dud22a] Andrej Dudenhefner. Certified decision procedures for two-counter machines. In Amy P. Felty, editor, *7th International Conference on Formal Structures for Computation and Deduction, FSCD 2022, August 2-5, 2022, Haifa, Israel*, volume 228 of *LIPIcs*, pages 16:1–16:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.FSCD.2022.16.

[Dud22b] Andrej Dudenhefner. Constructive many-one reduction from the halting problem to semi-unification. In Florin Manea and Alex Simpson, editors, *30th EACSL Annual Conference on Computer Science Logic, CSL 2022, February 14-19, 2022, Göttingen, Germany (Virtual Conference)*, volume 216 of *LIPIcs*, pages 18:1–18:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.CSL.2022.18.

[FHS18] Yannick Forster, Edith Heiter, and Gert Smolka. Verification of PCP-related computational reductions in Coq. In Jeremy Avigad and Assia Mahboubi, editors, *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10895 of *Lecture Notes in Computer Science*, pages 253–269. Springer, 2018. doi:10.1007/978-3-319-94821-8_15.

[FK19] Yannick Forster and Fabian Kunze. A certifying extraction with time bounds from Coq to call-by-value lambda calculus. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA*, volume 141 of *LIPIcs*, pages 17:1–17:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ITP.2019.17.

[FKW20] Yannick Forster, Fabian Kunze, and Maximilian Wuttke. Verified programming of Turing machines in Coq. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 114–128. ACM, 2020. doi:10.1145/3372885.3373816.

[FL19] Yannick Forster and Dominique Larchey-Wendling. Certified undecidability of intuitionistic linear logic via binary stack machines and Minsky machines. In Assia Mahboubi and Magnus O. Myreen, editors, *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, pages 104–117. ACM, 2019. doi:10.1145/3293880.3294096.

[FLWD+20] Yannick Forster, Dominique Larchey-Wendling, Andrej Dudenhefner, Edith Heiter, Dominik Kirst, Fabian Kunze, Gert Smolka, Simon Spies, Dominik Wehr, and Maximilian Wuttke. A Coq library of undecidable problems. In *The Sixth International Workshop on Coq for Programming Languages (CoqPL 2020)*, 2020. URL: https://github.com/uds-psl/coq-library-undecidability.

[FRD00] Manuel Fähndrich, Jakob Rehof, and Manuvir Das. Scalable context-sensitive flow analysis using instantiation constraints. In Monica S. Lam, editor, *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000*, pages 253–263. ACM, 2000. doi:10.1145/349299.349332.

[Gir72] Jean-Yves Girard. Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur. Thèse d'État, Univ. Paris VII, 1972.

[Hen88] Fritz Henglein. Type inference and semi-unification. In Jérôme Chailloux, editor, *Proceedings of the 1988 ACM Conference on LISP and Functional Programming, LFP 1988, Snowbird, Utah, USA, July 25-27, 1988*, pages 184–197. ACM, 1988. doi:10.1145/62678.62701.

[Hen93] Fritz Henglein. Type inference with polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):253–289, 1993. doi:10.1145/169701.169692.

[Hoo66] Philip K. Hooper. The undecidability of the Turing machine immortality problem. *J. Symb. Log.*, 31(2):219–234, 1966. doi:10.2307/2269811.

[Jea12] Emmanuel Jeandel. On immortal configurations in Turing machines. In S. Barry Cooper, Anuj Dawar, and Benedikt Löwe, editors, *How the World Computes - Turing Centenary Conference and 8th Conference on Computability in Europe, CiE 2012, Cambridge, UK, June 18-23, 2012. Proceedings*, volume 7318 of *Lecture Notes in Computer Science*, pages 334–343. Springer, 2012. doi:10.1007/978-3-642-30870-3_34.

[JK93] Said Jahama and Assaf J. Kfoury. A general theory of semi-unification. Technical report, Boston University Computer Science Department, 1993.

[Kar07] Jarkko Kari. The tiling problem revisited (extended abstract). In Jérôme Olivier Durand-Lose and Maurice Margenstern, editors, *Machines, Computations, and Universality, 5th International Conference, MCU 2007, Orléans, France, September 10-13, 2007, Proceedings*, volume 4664 of *Lecture Notes in Computer Science*, pages 72–79. Springer, 2007. doi:10.1007/978-3-540-74593-8_6.

[KM89] Paris C. Kanellakis and John C. Mitchell. Polymorphic unification and ML typing. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, pages 105–115. ACM Press, 1989. doi:10.1145/75277.75286.

[KMNS88] Deepak Kapur, David R. Musser, Paliath Narendran, and Jonathan Stillman. Semi-unification. In Kesav V. Nori and Sanjeev Kumar, editors, *Foundations of Software Technology and Theoretical Computer Science, Eighth Conference, Pune, India, December 21-23, 1988, Proceedings*, volume 338 of *Lecture Notes in Computer Science*, pages 435–454. Springer, 1988. doi:10.1007/3-540-50517-2_95.

[KO08] Jarkko Kari and Nicolas Ollinger. Periodicity and immortality in reversible computing. In Edward Ochmanski and Jerzy Tyszkiewicz, editors, *Mathematical Foundations of Computer Science 2008, 33rd International Symposium, MFCS 2008, Torun, Poland, August 25-29, 2008, Proceedings*, volume 5162 of *Lecture Notes in Computer Science*, pages 419–430. Springer, 2008. doi:10.1007/978-3-540-85238-4_34.

[Kor96] Ivan Korec. Small universal register machines. *Theor. Comput. Sci.*, 168(2):267–301, 1996. doi:10.1016/S0304-3975(96)00080-1.

[KTU88] Assaf J. Kfoury, Jerzy Tiuryn, and Paweł Urzyczyn. A proper extension of ML with an effective type-assignment. In Jeanne Ferrante and Peter Mager, editors, *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, pages 58–69. ACM Press, 1988. doi:10.1145/73560.73565.

[KTU90a] Assaf J. Kfoury, Jerzy Tiuryn, and Paweł Urzyczyn. ML typability is dexptime-complete. In *CAAP '90*, pages 206–220, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

[KTU90b] Assaf J. Kfoury, Jerzy Tiuryn, and Paweł Urzyczyn. The undecidability of the semi-unification problem (preliminary report). In Harriet Ortiz, editor, *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 13-17, 1990, Baltimore, Maryland, USA*, pages 468–476. ACM, 1990. doi:10.1145/100216.100279.

[KTU93a] Assaf J. Kfoury, Jerzy Tiuryn, and Paweł Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):290–311, 1993. doi:10.1145/169701.169687.

[KTU93b] Assaf J. Kfoury, Jerzy Tiuryn, and Paweł Urzyczyn. The undecidability of the semi-unification problem. *Inf. Comput.*, 102(1):83–101, 1993. doi:10.1006/inco.1993.1003.

[KTU94] Assaf J. Kfoury, Jerzy Tiuryn, and Paweł Urzyczyn. An analysis of ML typability. *J. ACM*, 41(2):368–398, 1994. doi:10.1145/174652.174659.

[Lei89a] Hans Leiß. Polymorphic recursion and semi-unification. In Egon Börger, Hans Kleine Büning, and Michael M. Richter, editors, *CSL '89, 3rd Workshop on Computer Science Logic, Kaiserslautern, Germany, October 2-6, 1989, Proceedings*, volume 440 of *Lecture Notes in Computer Science*, pages 211–224. Springer, 1989. doi:10.1007/3-540-52753-2_41.

[Lei89b] Hans Leiß. Semi-unification and type inference for polymorphic recursion, 1989.

[LF22] Dominique Larchey-Wendling and Yannick Forster. Hilbert's tenth problem in Coq (extended version). *Log. Methods Comput. Sci.*, 18(1), 2022. doi:10.46298/lmcs-18(1:35)2022.

[LH91] Hans Leiß and Fritz Henglein. A decidable case of the semi-unification problem. In Andrzej Tarlecki, editor, *Mathematical Foundations of Computer Science 1991, 16th International Symposium, MFCS'91, Kazimierz Dolny, Poland, September 9-13, 1991, Proceedings*, volume 520 of *Lecture Notes in Computer Science*, pages 318–327. Springer, 1991. doi:10.1007/3-540-54345-7_75.

[Mai90] Harry G. Mairson. Deciding ML typability is complete for deterministic exponential time. In Frances E. Allen, editor, *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, USA, January 1990*, pages 382–401. ACM Press, 1990. doi:10.1145/96709.96748.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978. doi:10.1016/0022-0000(78)90014-4.

[Min67] Marvin L. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, 1967.

[MS96] Yuri V. Matiyasevich and Géraud Sénizergues. Decision problems for semi-Thue systems with a few rules. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996*, pages 523–531. IEEE Computer Society, 1996. doi:10.1109/LICS.1996.561469.

[Myc84] Alan Mycroft. Polymorphic type schemes and recursive definitions. In Manfred Paul and Bernard Robinet, editors, *International Symposium on Programming, 6th Colloquium, Toulouse, France, April 17-19, 1984, Proceedings*, volume 167 of *Lecture Notes in Computer Science*, pages 217–228. Springer, 1984. doi:10.1007/3-540-12925-1_41.

[Pos46] Emil L. Post. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52(4):264–268, 1946.

[Pur87] Paul Walton Purdom. Detecting looping simplifications. In Pierre Lescanne, editor, *Rewriting Techniques and Applications, 2nd International Conference, RTA-87, Bordeaux, France, May 25-27, 1987, Proceedings*, volume 256 of *Lecture Notes in Computer Science*, pages 54–61. Springer, 1987. doi:10.1007/3-540-17220-3_5.

[Rey74] John C. Reynolds. Towards a theory of type structure. In Bernard Robinet, editor, *Programming Symposium, Proceedings Colloque sur la Programmation, Paris, France, April 9-11, 1974*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423. Springer, 1974. doi:10.1007/3-540-06859-7_148.

[Rog87] Hartley Rogers. *Theory of Recursive Functions and Effective Computability (Reprint from 1967)*. MIT Press, 1987.

[Wel99] Joe B. Wells. Typability and type checking in system F are equivalent and undecidable. *Ann. Pure Appl. Log.*, 98(1-3):111–156, 1999. doi:10.1016/S0168-0072(98)00047-5.

[Woj99] Arkadiusz Wojna. Counter machines. *Inf. Process. Lett.*, 71(5-6):193–197, 1999. doi:10.1016/S0020-0190(99)00116-7.