Verification of PCP-Related Computational Reductions in Coq

Yannick Forster, Edith Heiter, and Gert Smolka

Saarland University, Saarbrücken, Germany
{forster,heiter,smolka}@ps.uni-saarland.de

Abstract. We formally verify several computational reductions concerning the Post correspondence problem (PCP) using the proof assistant Coq. Our verification includes a reduction of the halting problem for Turing machines to string rewriting, a reduction of string rewriting to PCP, and reductions of PCP to the intersection problem and the palindrome problem for context-free grammars.

Keywords: Post Correspondence Problem, String Rewriting, Context-free Grammars, Computational Reductions, Undecidability, Coq

1 Introduction

A problem P can be shown undecidable by giving an undecidable problem Q and a computable function reducing Q to P. There are well known reductions of the halting problem for Turing machines (TM) to the Post correspondence problem (PCP), and of PCP to the intersection problem for context-free grammars (CFI). We study these reductions in the formal setting of Coq's type theory [16] with the goal of providing elegant correctness proofs.

Given that the reduction of TM to PCP appears in textbooks [9,3,15] and in the standard curriculum for theoretical computer science, one would expect that rigorous correctness proofs can be found in the literature. To our surprise, this is not the case. Missing is the formulation of the inductive invariants enabling the necessary inductive proofs to go through. Speaking with the analogue of imperative programs, the correctness arguments in the literature argue about the correctness of programs with loops without stating and verifying loop invariants.

By inductive invariants we mean statements that are shown inductively and that generalise the obvious correctness statements one starts with. Every substantial formal correctness proof will involve the construction of suitable inductive invariants. Often it takes ingenuity to generalise a given correctness claim to one or several inductive invariants that can be shown inductively.

It took some effort to come up with the missing inductive invariants for the reductions leading from TM to PCP. Once we had the inductive invariants, we had rigorous and transparent proofs explaining the correctness of the reductions in a more satisfactory way than the correctness arguments we found in the literature.

J. Avigad and A. Mahboubi (Eds.): ITP 2018, LNCS 10895, pp. 253–269, 2018.
The final authenticated version is available online at https://doi.org/10.1007/978-3-319-94821-8_15

Reduction of problems is transitive. Given a reduction $P \preceq Q$ and a reduction $Q \preceq R$, we have a reduction $P \preceq R$. This way, complex reductions can be factorised into simpler reductions. Following ideas in the literature, we will establish the reduction chain

$$\mathrm{TM} \preceq \mathrm{SRH} \preceq \mathrm{SR} \preceq \mathrm{MPCP} \preceq \mathrm{PCP}$$

where TM is the halting problem of single-tape Turing machines, SRH is a generalisation of the halting problem for Turing machines, SR is the string rewriting problem, and MPCP is a modified version of PCP fixing a first card. The most interesting steps are $\mathrm{SR} \preceq \mathrm{MPCP}$ and $\mathrm{MPCP} \preceq \mathrm{PCP}$.

We also consider the intersection problem (CFI) and the palindrome problem (CFP) for a class of linear context-free grammars we call Post grammars. CFP asks whether a Post grammar generates a palindrome, and CFI asks whether for two Post grammars there exists a string generated by both grammars. We will verify reductions $\mathrm{PCP} \preceq \mathrm{CFI}$ and $\mathrm{PCP} \preceq \mathrm{CFP}$, thus showing that CFP and CFI are both undecidable.

Coq's type theory provides an ideal setting for the formalisation and verification of the reductions mentioned. The fact that all functions in Coq are total and computable makes the notion of computable reductions straightforward.

The correctness arguments coming with our approach are inherently constructive, which is verified by the underlying constructive type theory. The main inductive data types we use are numbers and lists, which conveniently provide for the representation of strings, rewriting systems, Post correspondence problems, and Post grammars.

The paper is accompanied by a Coq development covering all results of this paper. The definitions and statements in the paper are hyperlinked with their formalisations in the HTML presentation of the Coq development at `http://www.ps.uni-saarland.de/extras/PCP`.

Organisation

We start with the necessary formal definitions covering all reductions we consider in Section 2. We then present each of the six reductions and conclude with a discussion of the design choices underlying our formalisations. Sections 3 to 8 on the reductions are independent and can be read in any order.

We only give definitions for the problems and do not discuss the underlying intuitions, because all problems are covered in a typical introduction to theoretical computer science and the interested reader can refer to various textbooks providing good intuitions, e.g. [9,15,3].

Contribution

Our reduction functions follow the ideas in the literature. The main contributions of the paper are the formal correctness proofs for the reduction functions. Here some ingenuity and considerable elaboration of the informal arguments in the

literature were needed. As one would expect, the formal proofs heavily rely on inductive techniques. In contrast, the informal proof sketches in the literature do not introduce the necessary inductions (in fact, they don't even mention inductive proofs). To the best of our knowledge, the present paper is the first paper providing formal correctness proofs for basic reductions to and from PCP.

2 Definitions

Formalising problems and computable reductions in constructive type theory is straightforward. A *problem* consists of a type X and a unary predicate p on X, and a *reduction* of (X, p) to (Y, q) is a function $f : X \to Y$ such that $\forall x.\ px \leftrightarrow q(fx)$. Note that the usual requirement that f is total and computable can be dropped since it is satisfied by every function in a constructive type theory. We write $p \preceq q$ and say that p *reduces to* q if a reduction of (X, p) to (Y, q) exists.

Fact 1. *If $p \preceq q$ and $q \preceq r$, then $p \preceq r$.*

The basic inductive data structures we use are *numbers* ($n ::= 0 \mid Sn$) and *lists* ($L ::= [\,] \mid s :: L$). We write $L_1 +\!\!+ L_2$ for the *concatenation* of two lists, $\overline{L}$ for the reversal of a list, $[\,fs \mid s \in A\,]$ for a map over a list, and $[\,fs \mid s \in A \wedge ps\,]$ for a map and filter over a list. Moreover, we write $s \in L$ if s is a member of L, and $L_1 \subseteq L_2$ if every member of L_1 is a member of L_2.

A *string* is a list of symbols, and a *symbol* is a number. The letters x, y, z, u, and v range over strings, and the letters a, b, c range over symbols. We write xy for $x +\!\!+ y$ and ax for $a :: x$. We use ϵ to denote the empty string. A *palindrome* is a string x such that $x = \overline{x}$.

Fact 2. *$\overline{xy} = \overline{y}\,\overline{x}$ and $\overline{\overline{x}} = x$.*

Fact 3. *If $xay = uav$, $a \notin x$, and $a \notin u$, then $x = u$ and $y = v$.*

Proof. By induction on x. □

A *card* x/y or a *rule* x/y a is a pair (x, y) of two strings. When we call x/y a card we see x as the upper and y as the lower string of the card. When we call x/y a rule we see x as the left and y as the right side of the rule.

The letters A, B, C, P, R range over list of cards or rules.

2.1 Post Correspondence Problem

A *stack* is a list of cards. The *upper trace* A^1 and the *lower trace* A^2 of a stack A are strings defined as follows:

$$[\,]^1 := \epsilon \qquad [\,]^2 := \epsilon$$
$$(x/y :: A)^1 := x(A^1) \qquad (x/y :: A)^2 := y(A^2)$$

Note that A^1 is the concatenation of the upper strings of the cards in A, and that A^2 is the concatenation of the lower strings of the cards in A. We say that a stack A *matches* if $A^1 = A^2$ and a *match* is a matching stack. An example for a match is the list $A = [\epsilon/ab,\ a/c,\ bc/\epsilon]$, which satisfies $A^1 = A^2 = abc$.

We can now define the predicate for the *Post correspondence problem*:

$$\mathsf{PCP}\,(P) \ :=\ \exists A \subseteq P.\ A \neq [\,] \wedge A^1 = A^2$$

Note that $\mathsf{PCP}\,(P)$ holds iff there exists a nonempty match $A \subseteq P$. We then say that A is a *solution* of P. For instance,

$$P = [a/\epsilon,\ b/a,\ \epsilon/bb]$$

is solved by the match

$$A = [\epsilon/bb,\ b/a,\ b/a,\ a/\epsilon,\ a/\epsilon].$$

While it is essential that A is a list providing for order and duplicates, P may be thought of as a finite set of cards.

We now define the predicate for the *modified Post correspondence problem*:

$$\mathsf{MPCP}\,(x/y, P) \ :=\ \exists A \subseteq x/y :: P.\ xA^1 = yA^2$$

Informally, $\mathsf{MPCP}\,(x/y, P)$ is like $\mathsf{PCP}\,(x/y :: P)$ with the additional constraint that the solution for $x/y :: P$ starts with the first card x/y.

Note that in contrary to most text books we leave open whether x/y is an element of P and instead choose A as subset of $x/y :: P$. While this might first seem more complicated, it actually eases formalisation. Including x/y into P would require MPCP to be a predicate on arguments of the form $(P, x/y, H : x/y \in P)$, i.e. dependent pairs containing a proof.

2.2 String Rewriting

Given a list R of rules, we define *string rewriting* with two inductive predicates $x \succ_R y$ and $x \succ_R^* y$:

$$\frac{x/y \in R}{uxv \succ_R uyv} \qquad \frac{}{z \succ_R^* z} \qquad \frac{x \succ_R y \qquad y \succ_R^* z}{x \succ_R^* z}$$

Note that $\succ_R^*$ is the reflexive transitive closure of $\succ_R$, and that $x \succ_R y$ says that y can be obtained from x with a single rewriting step using a rule in R.

Fact 4. *The following hold:*

1. *If $x \succ_R^* y$ and $y \succ_R^* z$, then $x \succ_R^* z$.*
2. *If $x \succ_R^* y$, then $ux \succ_R^* uy$.*
3. *If $x \succ_R^* y$ and $R \subseteq P$, then $x \succ_P^* y$.*

Proof. By induction on $x \succ_R^* y$. □

Note that the induction lemma for string rewriting can be stated as

$$\forall z.\ Pz \to (\forall xy.\ x \succ_R y \to Py \to Px) \to \forall x.\ x \succ_R^* z \to Px.$$

This is stronger than the lemma Coq infers, because of the quantification over z on the outside. The quantification is crucial for many proofs that do induction on derivations $x \succ_R z$, and we use the lemma throughout the paper without explicitly mentioning it.

We define the predicates for the *string rewriting problem* and the *generalised halting problem* as follows:

$$\begin{aligned} \mathsf{SR}\,(R,x,y) &:= x \succ_R^* y \\ \mathsf{SRH}\,(R,x,a) &:= \exists y.\ x \succ_R^* y \land a \in y \end{aligned}$$

We call the second problem *generalised halting problem*, because it covers the halting problem for deterministic single-tape Turing machines, but also the halting problems for nondeterministic machines or for more exotic machines that e.g. have a one-way infinite tape or can read multiple symbols at a time.

We postpone the definition of Turing machines and of the halting problem TM to section 8.

2.3 Post Grammars

A *Post grammar* is a pair (R, a) of a list R of rules and a symbol a. Informally, a Post grammar (R, a) is a special case of a context-free grammar with a single nonterminal S and two rules $S \to xSy$ and $S \to xay$ for every rule $x/y \in R$, where $S \neq a$ and S does not occur in R. We define the *projection* $\sigma_a A$ of a list of rules A with a symbol a as follows:

$$\begin{aligned} \sigma_a[\,] &:= a \\ \sigma_a(x/y :: A) &:= x(\sigma_a A)y \end{aligned}$$

We say that a Post grammar (R, a) *generates* a string u if there exists a nonempty list $A \subseteq R$ such that $\sigma_a A = u$. We then say that A is a *derivation of* u in (R, a).

We can now define the predicates for the problems CFP and CFI:

$$\begin{aligned} \mathsf{CFP}\,(R,a) &:= \exists A \subseteq R.\ A \neq [\,] \land \sigma_a A = \overline{\sigma_a A} \\ \mathsf{CFI}\,(R_1,R_2,a) &:= \exists A_1 \subseteq R_1\ \exists A_2 \subseteq R_2. \\ &\qquad A_1 \neq [\,] \land A_2 \neq [\,] \land \sigma_a A_1 = \sigma_a A_2 \end{aligned}$$

Informally, $\mathsf{CFP}\,(R, a)$ holds iff the grammar (R, a) generates a palindrome, and $\mathsf{CFI}\,(R_1, R_2, a)$ holds iff there exists a string that is generated by both grammars (R_1, a) and (R_2, a). Note that as Post grammars are special cases of context-free grammars, the reduction of PCP to CFG and CFI can be trivially extended to reductions to the respective problems for context-free grammars. We prove this formally in the accompanying Coq development.

2.4 Alphabets

For some proofs it will be convenient to fix a finite set of symbols. We represent such sets as lists and speak of *alphabets*. The letter Σ ranges over alphabets. We say that an alphabet Σ *covers* a string, card, or stack if Σ contains every symbol occurring in the string, card, or stack. We may write $x \subseteq \Sigma$ to say that Σ covers x since both x and Σ are lists of symbols.

2.5 Freshness

At several points we will need to pick fresh symbols from an alphabet. Because we model symbols as natural numbers, a very simple definition of freshness suffices. We define a function fresh such that $\mathsf{fresh}\ \Sigma \notin \Sigma$ for an alphabet Σ as follows:

$$\begin{aligned} \mathsf{fresh}\ [\,] &= 0 \\ \mathsf{fresh}\ (a :: \Sigma) &= 1 + a + \mathsf{fresh}\ \Sigma \end{aligned}$$

fresh has the following characteristic property:

Lemma 5. *For all $a \in \Sigma$, $\mathsf{fresh}\ \Sigma > a$.*

Proof. By induction on Σ, with a generalised. □

The property is most useful when exploited in the following way:

Corollary 6. *For all $a \in \Sigma$, $\mathsf{fresh}\ \Sigma \neq a$.*

An alternative approach to this is to formalise alphabets explicitly as types Σ. This has the advantage that arbitrarily many fresh symbols can be introduced simultaneously using definitions like $\Gamma := \Sigma + X$, and symbols in Γ stemming from Σ can easily be shown different from fresh symbols stemming from X by inversion. However, this means that strings $x : \Sigma^*$ have to be explicitly embedded pointwise when used as strings of type Γ^*, which complicates proofs.

In general, both approaches have benefits and tradeoffs. Whenever proofs rely heavily on inversion (as e.g. our proofs in Section 8), the alternative approach is favorable. If proofs need the construction of many strings, as most of our proofs do, modelling symbols as natural numbers shortens proofs.

3 SRH to SR

We show that SRH (the generalised halting problem) reduces to SR (string rewriting). We start with the definition of the reduction function. Let R, x_0, and a_0 be given.

We fix an alphabet Σ covering R, x_0, and a_0. We now add rules to R that allow $x \succ_R^* a_0$ if $a_0 \in x$.

$$P := R +\!\!+ [\, a a_0 / a_0 \mid a \in \Sigma \,] +\!\!+ [\, a_0 a / a_0 \mid a \in \Sigma \,]$$

Lemma 7. *If $a_0 \in x \subseteq \Sigma$, then $x \succ_P^* a_0$.*

Proof. For all $y \subseteq \Sigma$, $a_0 y \succ_P^* a_0$ and $y a_0 \succ_P^* a_0$ follow by induction on y. The claim now follows with Fact 4 (1,2). □

Lemma 8. $\mathsf{SRH}(R, x_0, a_0) \leftrightarrow \mathsf{SR}(P, x_0, a_0)$.

Proof. Let $x_0 \succ_R^* y$ and $a_0 \in y$. Then $y \succ_P^* a_0$ by Lemma 7. Moreover, $x_0 \succ_P^* y$ by Fact 4 (3). Thus $x_0 \succ_P^* a_0$ by Fact 4 (1).

Let $x_0 \succ_P^* a_0$. By induction on $x_0 \succ_P^* a_0$ it follows that there exists y such that $x_0 \succ_R^* y$ and $a_0 \in y$. □

Theorem 9. *SRH reduces to SR.*

Proof. Follows with Lemma 8. □

4 SR to MPCP

We show that SR (string rewriting) reduces to MPCP (the modified Post correspondence problem). We start with the definition of the reduction function.

Let R, x_0 and y_0 be given. We fix an alphabet Σ covering R, x_0, and y_0. We also fix two symbols $\$, \# \notin \Sigma$ and define:

$$\begin{aligned} d &:= \$ \,/\, \$x_0\# \\ e &:= y_0\#\$ \,/\, \$ \\ P &:= [d, e] +\!\!+ R +\!\!+ [\#/\#] +\!\!+ [\, a/a \mid a \in \Sigma \,] \end{aligned}$$

The idea of the reduction is as follows: Assume $\Sigma = [a, b, c]$ and rules bc/a and aa/b in R. Then $abc \succ_R aa \succ_R b$ and we have $d = \$/\$abc\#$, $e = b\#\$/\$$, and $P = [d, e, bc/a, aa/b, \ldots, a/a, b/b, c/c]$, omitting possibe further rules in R. Written suggestively, the following stack matches:

$\frac{\$}{\$abc\#}$	$\frac{a}{a}$	$\frac{bc}{a}$	$\frac{\#}{\#}$	$\frac{aa}{b}$	$\frac{\#}{\#}$	$\frac{b\#\$}{\$}$

And, vice versa, every matching stack starting with d will yield a derivation of $abc \succ_R^* b$.

We now go back to the general case and state the correctness lemma for the reduction function.

Lemma 10. *$x_0 \succ_R^* y_0$ if and only if there exists a stack $A \subseteq P$ such that $d :: A$ matches.*

From this lemma we immediately obtain the reduction theorem (Theorem 13). The proof of the lemma consists of two *translation lemmas*: Lemma 11 and Lemma 12. The translation lemmas generalise the two directions of Lemma 10 such that they can be shown with canonical inductions.

Lemma 11. *Let $x \subseteq \Sigma$ and $x \succ_R^* y_0$. Then there exists $A \subseteq P$ such that $A^1 = x\#A^2$.*

Proof. By induction on $x \succ_R^* y_0$. In the first case, $x = y_0$ and $[e]^1 = x\#[e]^2$. In the second case, $x \succ y$ and $y \succ^* y_0$. By induction hypothesis there is $A \subseteq P$ such that $A^1 = y\#A^2$. Let $x = (a_1 \dots a_n)u(b_1 \dots b_n)$ and $y = (a_1 \dots a_n)v(b_1 \dots b_n)$ for $u/v \in R$. We define $B := (a_1/a_1) \dots (a_n/a_n) :: (u/v) :: (b_1/b_1) \dots (b_n/b_n) :: (\#/\#) :: A$. Now $B^1 = x\#A^1 = x\#y\#A^2 = x\#B^2$. □

Lemma 12. *Let $A \subseteq P$, $A^1 = x\#yA^2$, and $x, y \subseteq \Sigma$. Then $yx \succ_R^* y_0$.*

Proof. By induction on A with x and y generalised. We do all cases in detail:

- The cases where $A = [\,]$ or $A = d :: B$ are contradictory.
- Let $A = e :: B$. By assumption, $y_0\#\$B^1 = x\#y\$B^2$. Then $x = y_0$, $y = \epsilon$ and $yx = y_0 \succ_R^* y_0$.
- Let $A = u/v :: B$ for $u/v \in R$. Because $\#$ is not in u and by assumption $uB^1 = x\#yvB^2$, $x = u +\!\!+ x'$. And $yx = yux' \succ yvx' \succ^* y_0$ by induction hypothesis.
- Let $A = \#/\# :: B$. By assumption, $\#B^1 = x\#y\#B^2$. Then $x = \epsilon$ and we have $B^1 = y\#\epsilon B^2$. By induction hypothesis, this yields $yx = \epsilon y \succ_R^* y_0$ as needed.
- Let $A = a/a :: B$ for $a \in \Sigma$ and assume $aB^1 = x\#yaB^2$. Then $x = ax'$ and $B^1 = x'\#yaB^2$. By induction hypothesis, this yields $yx = yax' \succ_R^* y_0$ as needed.

□

Theorem 13. *SR reduces to MPCP.*

Proof. Follows with Lemma 10. □

The translation lemmas formulate what we call the *inductive invariants* of the reduction function. The challenge of proving the correctness of the reduction function is finding strong enough inductive invariants that can be verified with canonical inductions.

5 MPCP to PCP

We show that MPCP (modified PCP) reduces to PCP.

The idea of the reduction is that for a stack $A = [x_1/y_1, \dots, x_n, y_n]$ and a first card x_0/y_0 where $x_i = a_i^0 \dots a_i^{m_i}$ and $y_i = b_i^0 \dots b_i^{m_i'}$ we have

$$\begin{aligned}&(a_0^0 \dots a_0^{m_0})(a_1^0 \dots a_1^{m_1}) \dots (a_n^0 \dots a_n^{m_n})\\ =&(b_0^0 \dots b_0^{m_0'})(b_1^0 \dots b_1^{m_1'}) \dots (b_n^0 \dots b_n^{m_n'})\end{aligned}$$

if and only if we have

$$\begin{aligned}&\$(\#a_0^0 \dots \#a_0^{m_0})(\#a_1^0 \dots \#a_1^{m_1}) \dots (\#a_n^0 \dots \#a_n^{m_n})\#\$\\ =&\$\#(b_0^0\# \dots b_0^{m_0'}\#)(b_1^0\# \dots b_1^{m_1'}\#) \dots (b_n^0\# \dots b_n^{m_n'}\#)\$.\end{aligned}$$

The reduction function implements this idea by constructing a dedicated first and a dedicated last card and by inserting #-symbols into the MPCP cards:

Let x_0/y_0 and R be given. We fix an alphabet Σ covering x_0/y_0 and R. We also fix two symbols $\$, \# \notin \Sigma$. We define two functions ${}^{\#}x$ and $x^{\#}$ inserting the symbol # before and after every symbol of a string x:

$$\begin{array}{rlrl} {}^{\#}\epsilon &:= \epsilon & \epsilon^{\#} &:= \epsilon \\ {}^{\#}(ax) &:= \#a({}^{\#}x) & (ax)^{\#} &:= a\#(x^{\#}) \end{array}$$

We define:

$$\begin{array}{rcl} d & := & \$({}^{\#}x_0) \,/\, \$\#(y_0^{\#}) \\ e & := & \#\$ \,/\, \$ \\ P & := & [d, e] +\!\!+ [\ {}^{\#}x \,/\, y^{\#} \mid x/y \in x_0/y_0 :: R \wedge (x/y) \neq (\epsilon/\epsilon)\ ] \end{array}$$

We now state the correctness lemma for the reduction function.

Lemma 14. *There exists a stack $A \subseteq x_0/y_0 :: R$ such that $x_0A^1 = y_0A^2$ if and only if there exists a nonempty stack $B \subseteq P$ such that $B^1 = B^2$.*

From this lemma we immediately obtain the desired reduction theorem (Theorem 19). The proof of the lemma consists of two translation lemmas (Lemmas 17 and 18) and a further auxiliary lemma (Lemma 15).

Lemma 15. *Every nonempty match $B \subseteq P$ starts with d.*

Proof. Let B be a nonempty match $B \subseteq P$. Then e cannot be the first card of B since the upper string and lower string of e start with different symbols. For the same reason ${}^{\#}x \,/\, y^{\#}$ cannot be the first card of B if $x/y \in R$ and both x and y are nonempty.

Consider $\epsilon/ay \in R$. Then $\epsilon/(ay)^{\#}$ cannot be the first card of B since no card of P has an upper string starting with a.

Consider $ax/\epsilon \in R$. Then ${}^{\#}(ay)/\epsilon$ cannot be the first card of B since no card of P has a lower string starting with #. □

For the proofs of the translation lemmas we need a few facts about ${}^{\#}x$ and $x^{\#}$.

Lemma 16. *The following hold:*

1. $({}^{\#}x)\# = \#(x^{\#})$.
2. ${}^{\#}(xy) = ({}^{\#}x)({}^{\#}y)$.
3. $(xy)^{\#} = (x^{\#})(y^{\#})$.
4. ${}^{\#}x \neq \#(y^{\#})$.
5. $x^{\#} = y^{\#} \to x = y$.

Proof. By induction on x. □

Lemma 17. *Let $A \subseteq x_0/y_0 :: R$ and $xA^1 = yA^2$. Then there exists a stack $B \subseteq P$ such that $(\#x)B^1 = \#(y\#)B^2$.*

Proof. By induction on A with x and y generalised. The case for $A = [\,]$ follows from Lemma 16 (1) by choosing $[e]$.

For the other case, let $A = x'/y' :: A'$. Then by assumption $xx'A'^1 = yy'A'^2$. And thus by induction hypothesis there exists $B \subseteq P$ such that $\#(xx')B^1 = \#(yy')\#B^2$. By Lemma 16 (2) and (3), $(\#x)(\#x')B^1 = \#(y\#)(y'\#)B^2$.

If $(x'/y') \neq (\epsilon/\epsilon)$, then choosing $\#x'/y'\# :: B \subseteq P$ works. Otherwise, $B \subseteq P$ works. □

Lemma 18. *Let $B \subseteq P$ such that $(\#x)B^1 = \#(y\#)B^2$ and $x, y \subseteq \Sigma$. Then there exists a stack $A \subseteq x_0/y_0 :: R$ such that $xA^1 = yA^2$.*

Proof. By induction on B. The cases $B = [\,]$ and $B = d :: B'$ yield contradictions using Lemma 16 (4). For $B = e :: B'$, choosing $A = [\,]$ works by Lemma 16 (5).

The interesting case is $B =^{\#} x'/y'\# :: B'$ for $x'/y' \in x_0/y_0 :: R$ with $(x'/y') \neq (\epsilon/\epsilon)$. By assumption and Lemma 16 (2) and (3) we know that $\#(xx')B'^1 = \#(yy')\#B'^2$. Now by induction hypothesis, where all premises follow easily, there is $A \subseteq x_0/y_0 :: R$ with $xx'A^1 = yy'A^2$ and thus $x'/y' :: A$ works. □

Theorem 19. *MPCP reduces to PCP.*

Proof. Follows with Lemma 14. □

6 PCP to CFP

We show that PCP reduces to CFP (the palindrome problem for Post grammars).

Let # be a symbol.

Fact 20. *Let $\# \notin x, y$. Then $x\#y$ is a palindrome iff $y = \overline{x}$.*

Proof. Follows with Facts 2 and 3. □

There is an obvious connection between matching stacks and palindromes: A stack

$$[x_1/y_1, \ldots, x_n/y_n]$$

matches if and only if the string

$$x_1 \cdots x_n \# \overline{y_n} \cdots \overline{y_1}$$

is a palindrome, provided the symbol # does not appear in the stack (follows with Facts 2 and 20 using $\overline{y_n} \cdots \overline{y_1} = \overline{y_1 \cdots y_n}$). Moreover, strings of the form $x_1 \cdots x_n \# \overline{y_n} \cdots \overline{y_1}$ with $n \geq 1$ may be generated by a Post grammar having a rule $x/\overline{y}$ for every card x/y in the stack. The observations yield a reduction of PCP to CFP.

We formalise the observations with a function

$$\gamma A := [\, x/\overline{y} \mid x/y \in A \,].$$

Lemma 21. $\sigma_{\#}(\gamma A) = A^1 \,\#\, \overline{A^2}$.

Proof. By induction on A using Fact 2. □

Lemma 22. *Let A be a stack and $\#$ be a symbol not occurring in A. Then A is a match if and only if $\sigma_{\#}(\gamma A)$ is a palindrome.*

Proof. Follows with Lemma 21 and Facts 20 and 2. □

Lemma 23. $\gamma(\gamma A) = A$ *and* $A \subseteq \gamma B \to \gamma A \subseteq B$.

Proof. By induction on A using Fact 2. □

Theorem 24. *PCP reduces to CFP.*

Proof. Let P be a list of cards. We fix a symbol $\#$ that is not in P and show $\mathsf{PCP}\,(P) \leftrightarrow \mathsf{CFP}\,(\gamma P, \#)$.

Let $A \subseteq P$ be a nonempty match. It suffices to show that $\gamma A \subseteq \gamma P$ and $\sigma_{\#}(\gamma A)$ is a palindrome. The first claim follows with Lemma 23, and the second claim follows with Lemma 22.

Let $B \subseteq \gamma P$ be a nonempty stack such that $\sigma_{\#} B$ is a palindrome. By Lemma 23 we have $\gamma B \subseteq P$ and $B = \gamma(\gamma B)$. Since γB matches by Lemma 22, we have $\mathsf{PCP}\,(P)$. □

7 PCP to CFI

We show that PCP reduces to CFI (the intersection problem for Post grammars). The basic idea is that a stack $A = [x_1/y_1, \ldots, x_n/y_n]$ with $n \geq 1$ matches if and only if the string

$$x_1 \cdots x_n \# \, x_n \# y_n \# \cdots \# x_1 \# y_1 \#$$

equals the string

$$y_1 \cdots y_n \# \, x_n \# y_n \# \cdots \# x_1 \# y_1 \#$$

provided the symbol $\#$ does not occur in A. Moreover, strings of these forms can be generated by the Post grammars $([\, x/x\#y\# \mid x/y \in A\,], \#)$ and $([\, y/x\#y\# \mid x/y \in A\,], \#)$, respectively.

We fix a symbol $\#$ and formalise the observations with two functions

$$\gamma_1 A := [\, x/x\#y\# \mid x/y \in A\,] \qquad \gamma_2 A := [\, y/x\#y\# \mid x/y \in A\,]$$

and a function γA defined as follows:

$$\gamma[\,] := [\,]$$
$$\gamma(x/y :: A) := (\gamma A) x \# y \#$$

Lemma 25. $\sigma_{\#}(\gamma_1 A) = A^1 \# \, (\gamma A)$ *and* $\sigma_{\#}(\gamma_2 A) = A^2 \# \, (\gamma A)$.

Proof. By induction on A. □

Lemma 26. *Let $B \subseteq \gamma_i\, C$. Then there exists $A \subseteq C$ such that $\gamma_i\, A = B$.*

Proof. By induction on B using Fact 3. □

Lemma 27. *Let # not occur in A_1 and A_2. Then $\gamma A_1 = \gamma A_2$ implies $A_1 = A_2$.*

Proof. By induction on A_1 using Fact 3. □

Theorem 28. *PCP reduces to CFI.*

Proof. Let P be a list of cards. We fix a symbol # not occurring in P and define $R_1 := \gamma_1 P$ and $R_2 := \gamma_2 P$. We show $\mathsf{PCP}\,(P) \leftrightarrow \mathsf{CFI}\,(R_1, R_2, \#)$.

Let $A \subseteq P$ be a nonempty match. Then $\gamma_1 A \subseteq R_1$, $\gamma_2 A \subseteq R_2$, and $\sigma_\#(\gamma_1 A) = \sigma_\#(\gamma_2 A)$ by Lemma 25.

Let $B_1 \subseteq R_1$ and $B_2 \subseteq R_2$ be nonempty lists such that $\sigma_\# B_1 = \sigma_\# B_2$. By Lemma 26 there exist nonempty stacks $A_1, A_2 \subseteq P$ such that $\gamma_i(A_i) = B_i$. By Lemma 25 we have $A_1^1 \#(\gamma A_1) = A_2^2 \#(\gamma A_2)$. By Fact 3 we have $A_1^1 = A_2^2$ and $\gamma A_1 = \gamma A_2$. Thus $A_1 = A_2$ by Lemma 27. Hence $A_1 \subseteq P$ is a nonempty match. □

Hopcroft et al. [9] give a reduction of PCP to CFI by using grammars equivalent to the following Post grammars:

$$\gamma_1 A := [\, x/i \mid x/y \in A \text{ at position } i \,] \quad \gamma_2 A := [\, y/i \mid x/y \in A \text{ at position } i \,]$$

While being in line with the presentation of PCP with indices, it complicates both the formal definition and the verification.

Hesselink [8] directly reduces CFP to CFI for general context-free grammars, making the reduction PCP to CFI redundant. The idea is that a context-free grammar over Σ contains a palindrome if and only if its intersection with the context-free grammar of all palindromes over Σ is non-empty. We give a formal proof of this statement using a definition of context-free rewriting with explicit alphabets.

For Post grammars, CFP is not reducible to CFI, because the language of all palindromes is not expressible by a Post grammar.

8 TM to SRH

A Turing machine, independent from its concrete type-theoretic definition, always consists of an alphabet Σ, a finite collection of states Q, an initial state q_0, a collection of halting states $H \subseteq Q$, and a step function which controls the behaviour of the head on the tape. The halting problem for Turing machines TM then asks whether a Turing machine M reaches a final state when executed on a tape containing a string x.

In this section, we briefly report on our formalisation of a reduction from TM to SRH following ideas from Hopcroft et al. [9]. In contrast to the other sections, we omit the technical details of the proof, because there are abundantly many,

and none of them is interesting from a mathematical standpoint. We refer the interested reader to [7] for all details.

In the development, we use a formal definition of Turing machines from Asperti and Ricciotti [1].

To reduce TM to SRH, a representation of configurations c of Turing machines as strings $\langle c \rangle$ is needed. Although the content of a tape can get arbitrarily big over the run of a machine, it is finite in every single configuration. It thus suffices to represent only the part of the tape that the machine has previously written to.

We write the current state to the left of the currently read symbol and, following [1], distinguish four non-overlapping situations: The tape is empty ($q(\!|\!)$), the tape contains symbols and the head reads one of them ($(\!|xqay|\!)$), the tape contains symbols and the head reads none of them, because it is in a left-overflow position where no symbol has been written before ($q(\!|ax|\!)$) or the right-overflow counterpart of the latter situation ($(\!|xaq|\!)$). Note the usage of left and right markers to indicate the end of the previously written part.

The reduction from TM to SRH now works in three steps. Given a Turing machine M, one can define whether a configuration c' is reachable from a configuration c using its transition function [1,7]. First, we translate the transition function of the Turing machine into a string rewriting system using the translation scheme depicted in Table 1.

Table 1. Rewriting rules x/y in R if the machine according to its transition function in state q_1 continues in q_2 and reads, writes and moves as indicated. For example, if the transition function of the machine indicates that in state q_1 if symbol a is read, the machine proceeds to state q_2, writes nothing and moves to the left, we add the rule $(\!|q_1 a \,/\, q_2 (\!| a$ and rules $c q_1 a \,/\, q_2 c a$ for every c in the alphabet.

Read	Write	Move	x	y	x	y	x	y
$\bot$	$\bot$	L	$q_1 (\!\|$	$q_2 (\!\|$	$a\, q_1 \|\!)$	$q_2\, a \|\!)$		
$\bot$	$\bot$	N	$q_1 (\!\|$	$q_2 (\!\|$	$q_1 \|\!)$	$q_2 \|\!)$		
$\bot$	$\bot$	R	$q_1 (\!\|\!)$	$q_2 (\!\|\!)$	$q_1 \|\!)$	$q_2 \|\!)$	$q_1 (\!\| a$	$(\!\| q_1\, a$
$\bot$	$\lfloor b \rfloor$	L	$q_1 (\!\|$	$q_2 (\!\| b$	$a\, q_1 \|\!)$	$q_2\, a\, b \|\!)$		
$\bot$	$\lfloor b \rfloor$	N	$q_1 (\!\|$	$(\!\| q_2\, b$	$q_1 \|\!)$	$q_2\, b \|\!)$		
$\bot$	$\lfloor b \rfloor$	R	$q_1 (\!\|$	$(\!\| b\, q_2$	$q_1 \|\!)$	$b\, q_2 \|\!)$		
$\lfloor a \rfloor$	$\bot$	L	$(\!\| q_1\, a$	$q_2 (\!\| a$	$c\, q_1\, a$	$q_2\, c\, a$		
$\lfloor a \rfloor$	$\bot$	N	$q_1\, a$	$q_2\, a$				
$\lfloor a \rfloor$	$\bot$	R	$q_1\, a$	$a\, q_2$				
$\lfloor a \rfloor$	$\lfloor b \rfloor$	L	$(\!\| q_1\, a$	$q_2 (\!\| b$	$c\, q_1\, a$	$q_2\, c\, b$		
$\lfloor a \rfloor$	$\lfloor b \rfloor$	N	$q_1\, a$	$q_2\, b$				
$\lfloor a \rfloor$	$\lfloor b \rfloor$	R	$q_1\, a$	$b\, q_2$				

Lemma 29. *For all Turing machines M and configurations c and c' there is a SRS R such that $\langle c \rangle \succ_R^* \langle c' \rangle$ if and only if the configuration c' is reachable from the configuration c by the machine M.*

In the development, we first reduce to a version of string rewriting with explicit alphabets, and then reduce this version to string rewriting as defined before.

This proof is by far the longest in our development. In its essence, it is only a shift of representation, making explicit that transition functions encode a rewriting relation on configurations. The proof is mainly a big case distinction over all possible shapes of configurations of a machine, which leads to a combinatorial explosion and a vast amount of subcases. The proof does, however, not contain any surprises or insights.

Note that, although we work with deterministic machines in the Coq development, the translation scheme described in Table 1 also works for nondeterministic Turing machines.

The second step of the reduction is to incorporate the set of halting states H. We define an intermediate problem SRH', generalising the definition of SRH to strings:

$$\mathsf{SRH}'(R, x, z) := \exists y.\ x \succ_R^* y \land \exists a \in z.\ a \in y$$

Note that $\mathsf{SRH}(R, x, a) \leftrightarrow \mathsf{SRH}'(R, x, [a])$. TM can then easily be reduced to SRH':

Lemma 30. TM *reduces to* SRH'.

Proof. Given a Turing machine M and a string x, M accepts x if and only if $\mathsf{SRH}(R, q_0 \langle\!| x |\!\rangle, z)$, where R is the system from the last lemma, q_0 is the starting state of M and z is a string containing exactly all halting states of M. □

Third, we can reduce SRH' to SRH:

Lemma 31. SRH' *reduces to* SRH.

Proof. Given a SRS R, a string x and a string z, we first fix an alphabet Σ covering R and x, and a fresh symbol $\#$. We then have $\mathsf{SRH}'(R, x, z)$ if and only if $\mathsf{SRH}(R +\!\!+ [a/\# \mid a \in z], x, \#)$. □

All three steps combined yield:

Theorem 32. TM *reduces to* SRH.

9 Discussion

We have formalised and verified a number of computational reductions to and from the Post correspondence problem based on Coq's type theory. Our goal was to come up with a development as elegant as possible. Realising the design

presented in this paper in Coq yields an interesting exercise practising the verification of list-processing functions. If the intermediate lemmas are hidden and just the reductions and accompanying correctness statements are given, the exercise gains difficulty since the correctness proofs for the reductions $\mathsf{SR} \preceq \mathsf{MPCP} \preceq \mathsf{PCP}$ require the invention of general enough inductive invariants (Lemmas 11, 12, 17, 18). To our surprise, we could not find rigorous correctness proofs for the reductions $\mathsf{TM} \preceq \mathsf{SR} \preceq \mathsf{MPCP} \preceq \mathsf{PCP}$ in the literature (e.g, [9,3,15]). Teaching these reductions without rigorous correctness proofs in theoretical computer science classes seems bad practice. As the paper shows, elegant and rigorous correctness proofs using techniques generally applicable in program verification are available.

The ideas for the reductions $\mathsf{TM} \preceq \mathsf{SRH} \preceq \mathsf{SR} \preceq \mathsf{MPCP} \preceq \mathsf{PCP}$ are taken from Hopcroft et al. [9]. They give a monolithic reduction of the halting problem for Turing machines to MPCP. The decomposition $\mathsf{TM} \preceq \mathsf{SRH} \preceq \mathsf{SR} \preceq \mathsf{MPCP}$ is novel. Davis et al. [3] give a monolithic reduction $\mathsf{SR} \preceq \mathsf{PCP}$ based on different ideas. The idea for the reduction $\mathsf{PCP} \preceq \mathsf{CFP}$ is from Hesselink [8], and the idea for the reduction $\mathsf{PCP} \preceq \mathsf{CFI}$ appears in Hopcroft et al. [9].

There are several design choices we faced when formalising the material presented in this paper.

1. We decided to formalise PCP without making use of the positions of the cards in the list P. Most presentations in the literature (e.g., [9,15]) follow Post's original paper [13] in using positions (i.e., indices) rather than cards in matches. An exception is Davis et al. [3]. We think formulating PCP with positions is an unnecessary complication.
2. We decided to represent symbols as numbers rather than elements of finite types serving as alphabets. Working with implicit alphabets represented as lists rather than explicit alphabets represented as finite types saves bureaucracy.
3. We decided to work with Post grammars (inspired by Hesselink [8]) rather than general context-free grammars since Post grammars sharpen the result and enjoy a particularly simple formalisation. In the Coq development, we show that Post grammars are an instance of context-free grammars.

Furthermore, we decided to put the focus of this paper on the elegant reductions and not to cover Turing machines in detail. While being a wide-spread model of computation, even the concrete formal definition of Turing machines contains dozens of details, all of them not interesting from a mathematical perspective.

The Coq development verifying the results of sections 3 to 7 consists of about 850 lines of which about one third realises specifications. The reduction $\mathsf{SR} \preceq \mathsf{SRH}$ takes 70 lines, $\mathsf{SR} \preceq \mathsf{MPCP}$ takes 105 lines, $\mathsf{MPCP} \preceq \mathsf{PCP}$ takes 206 lines, $\mathsf{PCP} \preceq \mathsf{CFP}$ takes 60 lines, and $\mathsf{PCP} \preceq \mathsf{CFI}$ takes 107 lines. The reduction $\mathsf{TM} \preceq \mathsf{SRH}$ takes 610 lines, 230 of them specification, plus a definition of Turing machines taking 291 lines.

Future Work

Undecidability proofs for logics are often done by reductions from PCP or related tiling problems. We thus want to use our work as a stepping stone to build a library of reductions which can be used to verify more undecidability proofs. We want to reduce PCP to the halting problem of Minsky machines to prove the undecidability of intuitionistic linear logic [11]. Another possible step would be to reduce PCP to validity for first-order logic [2], following the reduction from e.g. [12]. Many other undecidability proofs are also done by direct reductions from PCP, like the intersection problem for two-way-automata [14], unification in third-order logic [10], typability in the $\lambda\Pi$-calculus [4], satisfiability for more applied logics like HyperLTL [5], or decision problems of first order theories [17].

In this paper, we gave reductions directly as functions in Coq instead of appealing to a concrete model of computation. Writing down concrete Turing machines computing the reductions is possible in principle, but would be very tedious and distract from the elegant arguments our proofs are based on.

In previous work [6] we studied an explicit model of computation based on a weak call-by-value calculus L in Coq. L would allow an implementation of all reduction functions without much overhead, which would also formally establish the computability of all reductions.

Moreover, it should be straightforward to reduce PCP to the termination problem for L. Reducing the termination problem of L to TM would take considerable effort. Together, the two reductions would close the loop and verify the computational equivalence of TM, SRH, SR, PCP, and the termination problem for L. Both reducing PCP to L and implementing all reductions in L is an exercise in the verification of deeply embedded functional programs, and orthogonal in the necessary methods to the work presented in this paper.

References

1. Andrea Asperti and Wilmer Ricciotti. A formalization of multi-tape Turing machines. *Theoretical Computer Science*, 603:23–42, 2015.
2. Alonzo Church. A note on the Entscheidungsproblem. *J. Symb. Log.*, 1(1):40–41, 1936.
3. Martin D. Davis, Ron Sigal, and Elaine J. Weyuker. *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science*. Academic Press, 2nd edition, 1994.
4. Gilles Dowek. The undecidability of typability in the lambda-pi-calculus. In *International Conference on Typed Lambda Calculi and Applications*, pages 139–145. Springer, 1993.
5. Bernd Finkbeiner and Christopher Hahn. Deciding hyperproperties. In *CONCUR 2016*, pages 13:1–13:14, 2016.
6. Yannick Forster and Gert Smolka. Weak call-by-value lambda calculus as a model of computation in Coq. In *ITP 2017*, pages 189–206. Springer, LNCS 10499, 2017.
7. Edith Heiter. Undecidability of the Post correspondence problem in Coq. Bachelor's Thesis, Saarland University, `https://www.ps.uni-saarland.de/~heiter/bachelor.php`, 2017.

8. Wim H. Hesselink. Post's correspondence problem and the undecidability of context-free intersection. Manuscript, University of Groningen, `http://wimhesselink.nl/pub/whh513.pdf`, 2015.
9. John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 3rd edition, 2006.
10. Gerard P. Huet. The undecidability of unification in third order logic. *Information and control*, 22(3):257–267, 1973.
11. Dominique Larchey-Wendling and Didier Galmiche. The undecidability of boolean BI through phase semantics. In *LICS 2010*, pages 140–149. IEEE, 2010.
12. Zohar Manna. *Mathematical theory of computation*. Dover Publications, Incorporated, 2003.
13. Emil L. Post. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52(4):264–268, 1946.
14. Michael O. Rabin and Dana Scott. Finite automata and their decision problems. *IBM journal of research and development*, 3(2):114–125, 1959.
15. Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, International edition, 2012.
16. The Coq Proof Assistant. `http://coq.inria.fr`, 2017.
17. Ralf Treinen. A new method for undecidability proofs of first order theories. *Journal of Symbolic Computation*, 14(5):437–457, 1992.