Certified Undecidability of Intuitionistic Linear Logic via Binary Stack Machines and Minsky Machines

Yannick Forster
Saarland University
Saarland Informatics Campus, Saarbrücken, Germany
forster@ps.uni-saarland.de

Dominique Larchey-Wendling
Université de Lorraine, CNRS, LORIA
Vandœuvre-lès-Nancy, France
dominique.larchey-wendling@loria.fr

Abstract

We formally prove the undecidability of entailment in intuitionistic linear logic in Coq. We reduce the Post correspondence problem (PCP) via binary stack machines and Minsky machines to intuitionistic linear logic. The reductions rely on several technically involved formalisations, amongst them a binary stack machine simulator for PCP, a verified low-level compiler for instruction-based languages and a soundness proof for intuitionistic linear logic with respect to trivial phase semantics. We exploit the computability of all functions definable in constructive type theory and thus do not have to rely on a concrete model of computation, enabling the reduction proofs to focus on correctness properties.

CCS Concepts • **Theory of computation → Models of computation**; **Linear logic**; **Type theory**.

Keywords Undecidability, many-one reduction, binary stack machines, Minsky machines, intuitionistic linear logic, low-level compiler, constructive type theory, Coq

ACM Reference Format:
Yannick Forster and Dominique Larchey-Wendling. 2019. Certified Undecidability of Intuitionistic Linear Logic via Binary Stack Machines and Minsky Machines. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '19), January 14–15, 2019, Cascais, Portugal.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3293880.3294096

1 Introduction

Undecidability of advanced problems is usually established using computable (many-one) reductions from a problem already known to be undecidable. Such reductions rely on many subtle details and could thus be a prime example for the use of interactive theorem provers to assist ongoing research. However, formalisations of undecidability proofs are not common in the literature. There are two main obstacles in our eyes: (1) proofs on paper mostly omit the invariants needed for the verification of the reduction; (2) they omit the computability proof, which amounts to the formal verification of a program in the chosen model of computation.

Constructive type theory, as implemented in the proof assistant Coq [37], provides a particularly convenient setting for decidability and undecidability proofs. Since every function definable in constructive type theory is computable, one can use a synthetic approach to computability. This approach makes an explicit model of computation and explicit computability proofs unnecessary, enabling the proofs to focus on the invariants needed for the reduction.

In this paper we contribute a formalised chain of reductions, starting at the Post correspondence problem (PCP), via binary PCP (BPCP), BPCP with indices (iBPCP), binary stack machines (BSM), Minsky machines (MM) and (elementary) intuitionistic linear logic (both elLL and ILL):

$$\mathsf{PCP} \preceq \mathsf{BPCP} \preceq \mathsf{iBPCP} \preceq \mathsf{BSM} \preceq \mathsf{MM} \preceq \mathsf{elLL} \preceq \mathsf{ILL}$$

Combined with the reduction Halt $\preceq$ PCP of [10], this yields a fully formalised reduction from the halting problem of Turing machines to entailment in ILL.

PCP is a problem over a finite set of cards, each having a string at the top and bottom. Such a PCP instance is solvable if there is a non-empty finite sequence of those cards with an equal upper and lower string. We follow [10] in giving a definition of PCP with countably many symbols. We reduce PCP to BPCP, where symbols are restricted to Booleans and to iBPCP, where the sequence of cards is represented by a sequence of their indices (natural numbers).

We describe a general framework for the semantics of instruction-based machine models and explain how it can be used for compositional reasoning. We then define binary stack machines as an instance of such instruction-based machines. BSMs have a fixed number of binary stacks and programs consist of consecutively indexed PUSH and POP instructions, the latter allowing conditional jumps based on the obtained Boolean or if the stack is empty. A BSM program terminates if it jumps to a non-existing index. We verify an iBPCP simulator — which is a BSM program terminating if and only if a given iBPCP instance is solvable —

CPP '19, January 14–15, 2019, Cascais, Portugal

ACM ISBN 978-1-4503-6222-1/19/01…$15.00
https://doi.org/10.1145/3293880.3294096

by enumerating all possible sequences of cards. The program size is linear in the size of the instance and always has more than 80 instructions. It consists of several independent sub-programs, making the verification crucially rely on compositional reasoning.

We define Minsky machines [31] with a fixed number of natural-number registers as another instance of instruction-based machines. MM-programs consist of INC and DEC operations, the latter allowing conditional jumps if the register is zero. We describe and verify a general compiler for instruction-based languages and use it to compile BSM programs to MM programs. The main challenge is to rearrange jumps using a linker. For the reduction we have to prove correctness of the compiler for both terminating and non-terminating programs, a notable difference to other verified compilers (e.g. [5, 27]).

We formalise intuitionistic linear logic with a sequent calculus S_{ILL}. We introduce the elementary fragment of ILL [25, 26] together with a goal directed sequent calculus G_{eILL} which is equivalent to S_{ILL} on this fragment: both define the same entailment predicate. We give a denotational semantics for ILL using trivial phase semantics based on commutative monoids and prove soundness of S_{ILL} for this semantics. Finally, we show that entailment in G_{eILL} is complete for trivial phase semantics, yielding a reduction of eILL to ILL. Our arguments slightly generalise those of the proof in [26] to formalise the reduction chain MM $\preceq$ eILL $\preceq$ ILL.

We conclude the paper by summarising our efforts to build a library of undecidable problems in Coq [13] and by giving an overview of related work. The library should contain both entry points for plugging new undecidability results and tools to mechanise the corresponding reductions. We discuss some ongoing additions to our library.

The axiom-free Coq formalisation of all the results in this paper is available online and the main lemmas and theorems in the pdf version of the paper are hyper-linked with the html version of the Coq source code:

https://uds-psl.github.io/ill-undecidability

2 Type Theory and Formal Undecidability

2.1 Definitions

We write $\mathbb{P}$ to denote the type of propositions. The basic inductive data structures we use are *natural numbers* ($n : \mathbb{N} ::= 0 \mid S\,n$), *Booleans* ($b : \mathbb{B} ::= 0 \mid 1$), the *option type* ($\mathbb{O}\,X ::= (x : X) \mid \star$) and *lists* ($l : \mathbb{L}\,X ::= [] \mid x :: l$). We write $l_1 +\!\!+ l_2$ for the *concatenation* of two lists, $|l|$ for the length of a list, $\bar{l}$ for the reversal of a list, and $[\,f\,s \mid s \in l\,]$ for a map over a list. We define $l[i] : \mathbb{O}\,X$ as the i-th element of l. 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.

We will overload the notation for lists and use it for vectors as well. Given a type X, we write the (dependent) type of vectors of length n as X^n. We denote by $\mathbb{F}_n$ the *finite type of n elements* ($\alpha, \beta : \mathbb{F}_n ::= 0 \mid \ldots \mid n-1$). If $v : X^n$, $x : X$ and $\alpha : \mathbb{F}_n$, we define $v[\alpha]$ as the α-th component of v and $v[\alpha := x]$ as the vector v updated by x on its α-th component.

2.2 Undecidability in Coq

A *decision problem* consists of a type X and a unary predicate $p : X \to \mathbb{P}$ on X. A problem (X, p) is *decidable* if there is a function $f : X \to \mathbb{B}$ s.t. $\forall x.\, p\,x \leftrightarrow f\,x = 1$, or equivalently, if there is a (dependent) *decider* of type $\forall x : X, \{p\,x\} + \{\neg p\,x\}$.

Being able to define the concept of decidability without referring to a model of computation is a feature of constructive type theory: due to the commitment of its designers to effective methods, all definable functions are computable. There has been a lot of work on proving problems to be decidable using this notion, see e.g. [4, 7, 8, 21, 24, 30, 35].

Note that since it is consistent in constructive type theory to assume that all problems are "decidable," i.e. by the axiom $\forall P : \mathbb{P},\ \{P\} + \{\neg P\}$, the logical negation of decidability of a problem p can never be actually proven.[1] However, in practice, the undecidability of a problem is rarely established directly anyway. Most proofs are by reduction from another undecidable problem, and ultimately most problems are thus proven undecidable by reduction from an initial undecidable problem. In most textbook presentations, it is the halting problem (Halt) which is seen as a *seed of undecidability*.

The idea to exploit reductions to prove the undecidability of problems is well-known. It was first used to formalise undecidability proofs in a proof assistant without referring to an explicit model of computation in [10]. They define (many-one) reductions from a problem (X, p) to (Y, q) as a (computable) function $f : X \to Y$ such that $\forall x,\ p\,x \leftrightarrow q\,(f\,x)$. We write $p \preceq q$ and say that p *reduces to* q if a reduction of (X, p) to (Y, q) exists. In the case of axiom-free Coq, all such reductions are inherently computable and thus, computability can be dropped from the definition. This coincides with usual practice in non-formalised papers where the computability of reductions is left out most of the time.

Actually, many-one reductions are only an instance of the more general notion of Turing-reductions (see e.g. [34]): (X, p) is Turing-reducible to (Y, q), written $p \preceq_T q$, if p is decidable given that q is decidable.

Lemma 2.1. *If $p \preceq q$ then $p \preceq_T q$.*

The structure of undecidability proofs can be comprehensively described using an inductive predicate undec p:

$$\frac{}{\text{undec Halt}} \qquad \frac{p \preceq_T q \qquad \text{undec } p}{\text{undec } q}$$

[1] This illustrates that adding axioms to Coq could render the assumption that all functions are computable invalid, thus we safely commit ourselves to axiom-free Coq.

The first rule establishes the halting problem for Turing machines as a seed of undecidability. By the second rule, undecidability can be transferred via (Turing) reductions.

Lemma 2.2. undec $(\lambda x.\, \neg\, p\, x)$ *implies* undec p.

Proof. If p is decidable then so is $\lambda x.\, \neg\, p\, x$. □

Lemma 2.3. undec p *if and only if* Halt $\preceq_T p$.

A problem that would be decidable and undecidable at the same time would entail an obviously unwanted statement:

Lemma 2.4. *If p is decidable and* undec p *then* Halt *is decidable.*

We rely on the assumptions that Halt in [10] is a faithful implementation of the halting problem and that functions in Coq are indeed computable. If the latter would not be the case, the earlier mentioned publications on decidability proofs would all lose their grounds.

The choice of Halt as the seed of undecidability is arbitrary: most problems that are universally accepted as undecidable would work. Since the halting problem Halt for a concrete Turing machine model has already been reduced to PCP in Coq [10], we can safely use PCP as an alternate seed of undecidability. This also explains why the reductions in this paper start at PCP. Remark that we use this inductive definition of undec only for illustrative purposes. For our proofs, we simply rely on many-one reductions.

3 Post Correspondence Problem

All definitions concerning strings will be parametrised over a type X of symbols. We will later use natural numbers as symbols for PCP and Booleans as symbols for BPCP. A *string* is a list of symbols. We write X^* if the symbol type is X. We call strings over Booleans *bitstrings*. 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 *card* 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. A *card set* (CS) is finite set of cards represented by a list (where the order and multiplicity do not matter). A *stack* is a sequence of cards also represented by a list (but the order and multiplicity matter). The letters A, B, C range over stacks and the letter R ranges over CS. We use the letter I to range over a list of natural numbers.

The *upper trace* A^1 and the *lower trace* A^2 of a stack A are strings defined by the following equations:

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

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 $A^1 = A^2$. A *match* is a matching stack.

We define the predicate for the *Post correspondence problem* over elements of type X as symbols as:

$$\mathrm{PCP}\,(R : \mathrm{CS}_X) \;:=\; \exists A \subseteq R.\; A \neq [] \wedge A^1 = A^2$$

Note that PCP(R) holds iff there exists a non-empty match $A \subseteq R$. If we do not specify the type X, we mean the special case $\mathrm{PCP}\,(R : \mathrm{CS}_{\mathbb{N}})$ where the symbols type is $\mathbb{N}$, used as the standard one in [10]. Here we also define the *binary Post correspondence problem* fixing Booleans as symbols: $\mathrm{BPCP}\,(R : \mathrm{CS}_{\mathbb{B}}) := \mathrm{PCP}\,(R : \mathrm{CS}_{\mathbb{B}})$.

Finally, we define a version of BPCP using explicit indices, which is easier to treat for low-level machines. We define functions I^k_R for lists of indices I and $k \in \{1, 2\}$.

$$[]^k_R := \epsilon \qquad (i :: I)^k_R := I^k_R \qquad (\text{for } R[i] \text{ undefined})$$
$$(i :: I)^1_R := x(I^1_R) \qquad (i :: I)^2_R := y(I^2_R) \qquad (\text{for } R[i] = x/y)$$

and define binary PCP with indices:

$$\mathrm{iBPCP}\,(R : \mathrm{CS}_{\mathbb{B}}) \;:=\; \exists I \neq [],\; (\forall i \in I,\; i < |R|) \wedge I^1_R = I^2_R$$

Note the equivalence when we reverse lists of indices:

Lemma 3.1. iBPCP R *iff* $\exists I \neq [],\; (\forall i \in I,\; i < |R|) \wedge \overline{I}^1_R = \overline{I}^2_R$.

3.1 PCP Reduces to BPCP

We define a function $f : \mathbb{N}^* \to \mathbb{B}^*$ by $f\,\epsilon := \epsilon$ and $f\,(nx) := 1^n 0\,(f\,x)$ and we extend f to cards and lists of cards by pointwise application.

Lemma 3.2. *The following hold for $k \in \{1, 2\}$:*

1. $(f\,A)^k = f(A^k)$ *and* 2. *If $A \subseteq C$ then $f\,A \subseteq f\,C$.*

We define the inverse function $g : \mathbb{B}^* \to \mathbb{N} \to \mathbb{N}^*$ with an auxiliary argument by $g\,\epsilon\,n := \epsilon$ and $g\,(1x)\,n := g\,x\,(1 + n)$ and $g\,(0x)\,n := n(g\,x\,0)$. We write $g\,x$ for $g\,x\,0$ and again extend it pointwise to cards and stacks.

Lemma 3.3. *If $k \in \{1, 2\}$ and $A \subseteq f\,B$ then*

1. $(g\,A)^k = g(A^k)$ *and* 2. $g\,A \subseteq B$.

Theorem 3.4. PCP $\preceq$ BPCP.

Proof. We prove that $\mathrm{PCP}\,R \leftrightarrow \mathrm{BPCP}(f\,R)$.

For the direction from left to right let $[] \neq A \subseteq R$ with $A^1 = A^2$ be given. Then $f\,A \subseteq f\,R$ by Lemma 3.2 (2), $f\,A \neq []$ is trivial and $(f\,A)^1 = (f\,A)^2$ follows from Lemma 3.2 (1).

For the direction from right to left let $[] \neq B \subseteq f\,R$ with $B^1 = B^2$ be given. $g\,B \subseteq R$ follows by Lemma 3.3 (2), $g\,B \neq []$ is trivial and $(g\,B)^1 = (g\,B)^2$ by Lemma 3.3 (1). □

3.2 BPCP Reduces to iBPCP

We define

$$f\,R\,[] := []$$
$$f\,R\,(x/y :: A) := n :: f\,R\,A \qquad (\text{if } x/y \in R \text{ at pos. } n)$$
$$f\,R\,(x/y :: A) := f\,R\,A \qquad (\text{if } x/y \notin R)$$

and

$$\begin{aligned} g\,R\,[] &:= [] \\ g\,R\,(i :: I) &:= (x/y)(g\,R\,I) && (\text{if } R[i] = x/y) \\ g\,R\,(i :: I) &:= g\,R\,I && (\text{if } R[i] \text{ is undefined}) \end{aligned}$$

Lemma 3.5. *The following hold for $k \in \{1, 2\}$:*

1. *If $B \subseteq R$ then $(f\,R\,B)^k_R = B^k$*
2. *If $(\forall i \in I,\ i < |R|)$ then $(g\,R\,I)^k = I^k_R$.*

Proof. (1) is by induction on B, (2) by induction on I. □

Theorem 3.6. BPCP $\preceq$ iBPCP.

4 Code

We define code over arbitrary instruction sets to reuse later for both binary stack machines and Minsky machines. Since both instruction sets contain absolute jump instructions, the positions of instructions are represented by natural numbers.

4.1 Syntax

We model the type of code type-theoretically by defining it over a type of instructions $\mathbb{I}$ as $\mathbb{N} \times \mathbb{L}\,\mathbb{I}$. We use the letter ρ for individual instructions and P for lists of instructions. A code $(i, [\rho_0; \rho_1; \ldots; \rho_{n-1}])$ can be seen as a labelled program:

$$i : \rho_0; \quad i+1 : \rho_1; \quad \ldots \quad i+n-1 : \rho_{n-1};$$

where $i, i+1, \ldots, i+n-1$ represent *program counter (PC)* values which identify the positions of individual instructions. The reason we use any number i instead of just 0 as the first label is to be able to reason about the whole code as well as about segments in the code, segments which we call *subcode*. We define the subcode relation denoted $(i, P) <_{\text{sc}} (j, Q)$ by

$$(i, P) <_{\text{sc}} (j, Q) := \exists\, l\, r,\ Q = l +\!\!+ P +\!\!+ r \wedge i = j + |l|$$

meaning that P is the segment of Q starting at label i. Many of the upcoming semantic relations below defined will be "somehow preserved" by the subcode relation and this is an essential ingredient of *compositional reasoning over subcode;* see Section 4.3 for more precise preservation properties.

The start label of a code (i, P) is $\text{start}(i, P) := i$. The end of code for (i, P) is the first label above the code block, i.e. $\text{end}(i, P) := i + |P|$. We define when a PC value k is in(side) or else out(side) of the code block (i, P) as

$$\begin{aligned} \text{in}\ k\ (i, P) &:= \text{start}(i, P) \le k < \text{end}(i, P) \\ \text{out}\ k\ (i, P) &:= k < \text{start}(i, P) \vee \text{end}(i, P) \le k. \end{aligned}$$

4.2 Semantics

The semantics of all our machines is defined by providing a type of states $\mathbb{S} := \mathbb{N} \times \mathbb{C}$ where a state $(i, v) : \mathbb{S}$ is composed of a PC value $i : \mathbb{N}$ and an abstract *configuration* $v : \mathbb{C}$,[2] and a step relation denoted $\rho \mathbin{/\!/} (i_1, v_1) \succ (i_2, v_2)$ of type $\mathbb{I} \to \mathbb{S} \to \mathbb{S} \to \mathbb{P}$ which describes the effect of instruction ρ, transforming the state (i_1, v_1) into the state (i_2, v_2).

We inductively extend the step relation to a small-step semantics of type $\mathbb{N} \times \mathbb{L}\,\mathbb{I} \to \mathbb{S} \to \mathbb{N} \to \mathbb{S} \to \mathbb{P}$ denoted by $(i, P) \mathbin{/\!/} (i_1, v_1) \succ^n (i_2, v_2)$, and meaning that the program (i, P), starting from state (i_1, v_1), reaches state (i_2, v_2) after the execution of n individual instructions:

$$\frac{}{(i, P) \mathbin{/\!/} (i_1, v_1) \succ^0 (i_1, v_1)}$$

$$\frac{P = P_1 +\!\!+ \rho :: P_2 \qquad i_1 = i + |P_1| \qquad \rho \mathbin{/\!/} (i_1, v_1) \succ (i_2, v_2) \qquad (i, P) \mathbin{/\!/} (i_2, v_2) \succ^n (i_3, v_3)}{(i, P) \mathbin{/\!/} (i_1, v_1) \succ^{n+1} (i_3, v_3)}$$

The first two premises of the second rule express that instruction ρ occurs at position i_1 in the program (i, P), or equivalently $(i_1, [\rho]) <_{\text{sc}} (i, P)$. Notice that whenever j satisfies out j (i, P) then $(i, P) \mathbin{/\!/} (j, v) \succ^n s$ implies $n = 0$ and $s = (j, v)$, i.e. the computation is blocked outside of (i, P).

To shorten notation, we will often use s to represent states like (i, v) and $\mathcal{P}$ to represent code like (i, P). We define the transitive closure $\succ^+$ and reflexive-transitive closure $\succ^*$ of the step relation as follows:

$$\begin{aligned} \mathcal{P} \mathbin{/\!/} s \succ^+ s' &:= \exists n > 0,\ \mathcal{P} \mathbin{/\!/} s \succ^n s' \\ \mathcal{P} \mathbin{/\!/} s \succ^* s' &:= \exists n,\ \mathcal{P} \mathbin{/\!/} s \succ^n s' \end{aligned}$$

We call the step relation *deterministic* if for any ρ and s, s_1 and s_2, if $\rho \mathbin{/\!/} s \succ s_1$ and $\rho \mathbin{/\!/} s \succ s_2$ then $s_1 = s_2$. In this paper, we only consider deterministic step relations. In this case, the small-step semantics is also deterministic, i.e. for any $\mathcal{P}$, s and n, there is at most one s' s.t. $\mathcal{P} \mathbin{/\!/} s \succ^n s'$. However, there are generally many possible s' s.t. $\mathcal{P} \mathbin{/\!/} s \succ^* s'$. Thus neither $\succ^*$ nor $\succ^+$ are a well-suited big-step semantics because they are not deterministic. By further assuming termination, we obtain the unicity of the end state.

Hence we define the (deterministic) big-step semantics as the *terminates on* predicate, and in addition, we also define the *terminates on in p steps* predicate, the *terminates on with progress* predicate and the *termination* predicate as:

$$\begin{aligned} \mathcal{P} \mathbin{/\!/} s \rightsquigarrow (j, w) &:= \mathcal{P} \mathbin{/\!/} s \succ^* (j, w) \wedge \text{out}\ j\ \mathcal{P} \\ \mathcal{P} \mathbin{/\!/} s \rightsquigarrow^n (j, w) &:= \mathcal{P} \mathbin{/\!/} s \succ^n (j, w) \wedge \text{out}\ j\ \mathcal{P} \\ \mathcal{P} \mathbin{/\!/} s \rightsquigarrow^+ (j, w) &:= \mathcal{P} \mathbin{/\!/} s \succ^+ (j, w) \wedge \text{out}\ j\ \mathcal{P} \\ \mathcal{P} \mathbin{/\!/} s \downarrow &:= \exists s',\ \mathcal{P} \mathbin{/\!/} s \rightsquigarrow s' \end{aligned}$$

Finally note that although out j $\mathcal{P}$ is a sufficient condition for j to be a state where the program cannot progress any more, this condition is not necessary. Indeed, in certain models of computation, some instructions might block a program, either unconditionally like a HALT instruction, or conditionally when a POP instruction is not allowed to run on an empty stack. To avoid having to distinguish between halting when outside or halting on a blocking instruction, we only manipulate machine models where the step relation is *total,* i.e. the property $\forall \rho\, s\, \exists s',\ \rho \mathbin{/\!/} s \succ s'$ holds. In that

[2] meaning that $\mathbb{C}$ is a type parameter to be instantiated later on, usually together with the type parameter $\mathbb{I}$ of instructions.

case, being outside of the code is the only way to block a computation.

4.3 Compositional Reasoning over Subcode

Compositional reasoning is the ability to derive properties of the whole from the properties of its components. It is an essential tool in the certification of big developments such as low-level computer code.

In our setting, compositional reasoning translates into tools which allow the transfer of properties of subcode to larger code. For instance, the following property is a simple and obvious example of compositional reasoning:

Lemma 4.1. *Given $\mathcal{P} <_{\text{sc}} \mathcal{Q}$ we have for any n, s, s':*

$$\mathcal{P} \mathbin{/\!/} s \succ^n s' \to \mathcal{Q} \mathbin{/\!/} s \succ^n s'$$

This reflects the fact that a computation running inside some subcode $\mathcal{P}$ of $\mathcal{Q}$ also runs inside $\mathcal{Q}$ itself. A more critical tool of compositional reasoning is the following result:

Lemma 4.2. *Given $\mathcal{P} <_{\text{sc}} \mathcal{Q}$ we have for any s_1, s_3:*

$$\mathcal{Q} \mathbin{/\!/} s_1 \rightsquigarrow s_3 \to \exists s_2, \mathcal{P} \mathbin{/\!/} s_1 \rightsquigarrow s_2 \wedge \mathcal{Q} \mathbin{/\!/} s_2 \rightsquigarrow s_3$$

Any terminating computation in code $\mathcal{Q}$ can be decomposed into a prefix terminating computation in subcode $\mathcal{P}$ followed by a suffix terminating computation in $\mathcal{Q}$.

Lemma 4.3. *If we assume the step relation $\succ$ to be deterministic, we get:*

$$\begin{array}{c}\mathcal{P} <_{\text{sc}} \mathcal{Q} \wedge \mathcal{P} \mathbin{/\!/} s_1 \succ^+ s_2 \wedge \mathcal{Q} \mathbin{/\!/} s_1 \rightsquigarrow^p s_3 \\ \to \exists q < p,\ \mathcal{Q} \mathbin{/\!/} s_2 \rightsquigarrow^q s_3\end{array}$$

This last result allows the analysis of terminating computations in $\mathcal{Q}$ by induction on their length, using the behaviour of its subcodes such as $\mathcal{P}$, as soon as they make the computation progress.

Such tools are our major examples of compositional reasoning: they are used to derive e.g. soundness and completeness properties of computations in code $\mathcal{Q}$ from the soundness properties of progressing computations in its subcodes.

5 Binary Stack Machines

We define *Binary Stack Machines (BSM)* as a particular instance of (low-level) code. BSM have a fixed number $n : \mathbb{N}$ of Boolean stacks modelled by lists of Booleans. We fix a number n of stacks as a parameter for all further definitions. BSM configurations are vectors v of n Boolean stacks, i.e. $\mathbb{C} := (\mathbb{L}\,\mathbb{B})^n$. We define BSM instructions as follows:

$$\mathbb{I}_n ::= \mathsf{POP}\ (\alpha : \mathbb{F}_n)\ (p : \mathbb{N})\ (q : \mathbb{N}) \mid \mathsf{PUSH}\ (\alpha : \mathbb{F}_n)\ (b : \mathbb{B}).$$

5.1 Semantics for BSM

The rules defining the step relation for BSM instructions are:

$$\begin{array}{ll}
\mathsf{PUSH}\ \alpha\ b \mathbin{/\!/} (i, v) \succ (1 + i, v[\alpha := b :: l]) & \text{if } v[\alpha] = l \\
\mathsf{POP}\ \alpha\ p\ q \mathbin{/\!/} (i, v) \succ (q, v) & \text{if } v[\alpha] = [] \\
\mathsf{POP}\ \alpha\ p\ q \mathbin{/\!/} (i, v) \succ (p, v[\alpha := l]) & \text{if } v[\alpha] = 0 :: l \\
\mathsf{POP}\ \alpha\ p\ q \mathbin{/\!/} (i, v) \succ (1 + i, v[\alpha := l]) & \text{if } v[\alpha] = 1 :: l
\end{array}$$

The $\mathsf{PUSH}\ \alpha\ b$ instruction pushes $b : \mathbb{B}$ to the stack α. The $\mathsf{POP}\ \alpha\ p\ q$ instruction (1) jumps to PC value q if the stack α is empty, or (2) jumps to p if the top element of the stack α is 0 and removes it or (3) increases the PC by one if the top element of the stack α is 1 and removes it.

The above defined step relation for BSM is obviously deterministic and total: individual instructions always execute and depending on the input state, lead to a unique output state. Finally, we define the halting problem for binary stack machines as a predicate over a dependent quadruplet:

$$\mathsf{BSM}\big(n : \mathbb{N}, i : \mathbb{N}, P : \mathbb{L}\,\mathbb{I}_n, v : (\mathbb{L}\,\mathbb{B})^n\big) := (i, P) \mathbin{/\!/} (i, v)\downarrow.$$

Note that the execution starts with the first instruction of the program and that we do not care about its final state as long as it terminates (on a PC value outside of its code).

5.2 Overview of the Reduction from iBPCP to BSM

We first describe the overall idea of the reduction and then explain some parts in detail. Recall that an iBPCP instance is given as a list $R : \mathbb{L}\,(\mathbb{L}\,\mathbb{B} \times \mathbb{L}\,\mathbb{B})$ of binary cards. The problem is to find a non-empty list of indices of cards of R such that the upper and lower trace are equal. We solve this problem using a simulator denoted $\mathtt{pcp\_bsm}_R$ implemented as a BSM-program in which the iBPCP instance R is hard-coded.

So let us fix the list R. The program $\mathtt{pcp\_bsm}_R$ will solve the instance generated by R. A card c_i in $R = [c_0; \ldots; c_i; \ldots]$ is represented by its index $i < |R|$. Finding a match is decomposed into three problems: (1) enumerate all the possible non-empty lists of card indices below $|R|$; (2) given a non-empty list of card indices, decode it into an upper trace and a lower trace and (3) compare those two traces for equality.

The simulator $\mathtt{pcp\_bsm}_R$ has 4 binary stacks s, u, l and a: stack s contains the encoding of the current list of cards indices, u and l contain the upper and lower traces, and a is a spare stack for internal computations:

(1) $\mathtt{pcp\_bsm}_R$ solves part (1) by generating in s all the possible non-empty finite binary sequences. We use a simple encoding of $\mathbb{L}\,\mathbb{N}$ in $\mathbb{L}\,\mathbb{B}$: e.g. the list $[5; 3; 0; 2]$ of card indices is to be recognised as the sequence $000001{\cdot}0001{\cdot}1{\cdot}001$ (the $\cdot$ only helps at reading). Any binary sequence which is not a valid encoding of cards — be it terminating with a 0 or be it containing indices larger than $|R|$ — is going to be rejected as invalid in part (2) and $\mathtt{pcp\_bsm}_R$ will branch again at (1) to generate the next possible binary sequence in s.

(2) Given an encoding of a non-empty sequence of cards indices in s, $\mathtt{pcp\_bsm}_R$ computes the upper (resp. lower) trace of that sequence in u (resp. l). To decode a card index at the top of s, it uses a dispatcher that analyses the head sequence of s of the shape $0^i1\ldots$ If i exceeds $|R|$ or s ends up empty before a 1 is encountered, then the dispatcher rejects the whole sequence s as an invalid encoding of card indices and jumps back to part (1). Otherwise, it decodes the cards indices one after the other and, when encountering the index i of card c_i, defers the execution into a hard coded version of card $c_i = (x_i/y_i)$ that pushes the upper part of x_i into u and the lower part y_i into l, i.e. $u \rightsquigarrow x_i +\!\!+ u$ and $l \rightsquigarrow y_i +\!\!+ l$ and then continues the analysis of the remaining part of s. Note that after the decoding of the sequences $[i_1;\ldots;i_k]$ of indices, we get $x_{i_k} +\!\!+ \cdots +\!\!+ x_{i_1}$ in u and $y_{i_k} +\!\!+ \cdots +\!\!+ y_{i_1}$ in l, i.e. the cards are stacked in reversed order, but we proved that this is no problem in Lemma 3.1.

(3) Given an upper/lower trace in u/l, $\mathtt{pcp\_bsm}_R$ solves part (3) by a 1-to-1 comparison between u and l. If u differs from l, $\mathtt{pcp\_bsm}_R$ reverts back to generating the next binary sequence by branching at (1). If u equals l then $\mathtt{pcp\_bsm}_R$ jumps to a terminating PC value outside of its code.

Notice that $\mathtt{pcp\_bsm}_R$ never stops if there is no solution to the iBPCP instance R. On the other hand, since (1) enumerates any possible card list (including some possibly invalid encodings which are rejected as such), if there is a solution to R, then $\mathtt{pcp\_bsm}_R$ will stop when (3) detects such a solution where $u = l$. Of course, these parts (1)-(3) correspond to specific BSM code for which correctness and completeness results are established. Here we only explain certain sub-programs of the $\mathtt{pcp\_bsm}_R$ program.

5.3 Details of the Reduction from iBPCP to BSM

To solve part (1), we enumerate all index lists using a function $\mathtt{next} : \mathbb{L}\,\mathbb{B} \to \mathbb{L}\,\mathbb{B}$ such that the sequence of its iterations starting at $[0]$ visits all non-empty binary lists:

Lemma 5.1. *There is a function* $\mathtt{next} : \mathbb{L}\,\mathbb{B} \to \mathbb{L}\,\mathbb{B}$ *such that*

$$\forall A : \mathbb{L}\,\mathbb{B},\ A \neq [] \to \exists n : \mathbb{N},\ \mathtt{next}^n\,[0] = A.$$

Then we show that next is BSM-computable:

Lemma 5.2. *For any* $n, i : \mathbb{N}$ *and* $x \neq y : \mathbb{F}_n$ *there is a program* $\mathtt{increment} : \mathbb{L}\,\mathbb{I}_n$ *with 15 instructions satisfying*

$$\forall v,\ (i, \mathtt{increment}) \mathbin{/\!/} (i, v) \succ^* (15 + i, v[x := \mathtt{next}\,(v[x])]).$$

Notice the absence of y in the specification of `increment` witnessing its use as a spare register: its content might be locally modified during the execution of `increment` but it is reverted back to its initial value afterwards. Using a simple loop over `increment` allows us to solve part (1) of $\mathtt{pcp\_bsm}_R$.

To solve part (2) we implement programs $\mathtt{tile}_{\alpha/\beta}$ pushing the strings α and β to the stacks x and y:

Lemma 5.3. *For all* $n : \mathbb{N}, x \neq y : \mathbb{F}_n$ *and* $\alpha, \beta : \mathbb{L}\,\mathbb{B}$ *there is a program* $\mathtt{tile}_{\alpha/\beta} : \mathbb{L}\,\mathbb{I}_n$ *with* $|\mathtt{tile}_{\alpha/\beta}| = |\alpha| + |\beta|$ *satisfying the following specification:*

$$\forall v\, i\, w,\ \ w = v\big[x := \alpha +\!\!+ v[x], y := \beta +\!\!+ v[y]\big] \\ \to (i, \mathtt{tile}_{\alpha/\beta}) \mathbin{/\!/} (i, v) \succ^* (|\mathtt{tile}_{\alpha/\beta}| + i, w).$$

The cards decoder consists in gluing $\mathtt{tile}_{c_1}$, $\mathtt{tile}_{c_2}$, …, $\mathtt{tile}_{c_k}$ where $R = [c_1;\ldots;c_k]$, interleaved with a measure of the number j of head 0's in $v[s] = 0^j1\ldots$ to branch on the appropriate $\mathtt{tile}_{c_j}$. This decoders also detects and rejects a value $j \geq |R|$ or a stack $v[s]$ of the form 0^* not containing any 1. This card index decoder is implemented by structural induction on R. We refer to the Coq term `full_decoder` in file `bsm_utils.v` for more detailed explanations. This solves part (2) of $\mathtt{pcp\_bsm}_R$.

To solve part (3), we implement stack comparison.

Lemma 5.4. *For all* $n : \mathbb{N}, x \neq y : \mathbb{F}_n$ *and* $i, p, q : \mathbb{N}$*, there is a program* $\mathtt{compare\_stacks} : \mathbb{L}\,\mathbb{I}_n$ *(of length 10) which satisfies the following specification:*

$$\forall v\, \exists j\, w,\ \ (i, \mathtt{compare\_stacks}) \mathbin{/\!/} (i, v) \succ^* (j, w) \\ \wedge\ \forall z, z \neq x \to z \neq y \to v[z] = w[z] \\ \wedge\ (v[x] = v[y] \wedge j = p \vee v[x] \neq v[y] \wedge j = q).$$

Hence, `compare_stacks` can distinguish between $v[x] = v[y]$ and $v[x] \neq v[y]$ by jumping to either p or (another) PC value q. Notice that the resulting state w is not affected too much by the computation since it preserves any stack register except x and y. Since the values of $w[x]$ and $w[y]$ are unspecified, we might have to erase them afterwards:

Lemma 5.5. *The program* $(i, \mathtt{empty\_stack})$ *with*

$$\mathtt{empty\_stack} := [\mathtt{POP}\ x\ i\ (3 + i); \mathtt{PUSH}\ x\ 0; \mathtt{POP}\ x\ i\ i]$$

composed of 3 instructions satisfies the specification

$$\forall v,\ \ (i, \mathtt{empty\_stack}) \mathbin{/\!/} (i, v) \succ^* (3 + i, v[x := []]).$$

This solves part (3) of $\mathtt{pcp\_bsm}_R$. To complete the reduction of an iBPCP instance to a BSM instance we have to prove soundness and completeness of the $\mathtt{pcp\_bsm}_R$ program. We do this by combining the previous results using the tools of compositional reasoning described in Section 4.3.

Theorem 5.6 (iBPCP simulator). *For any configuration* v *and any state* s*, the two following properties hold:*

$$\text{iBPCP}\ R \to (1, \mathtt{pcp\_bsm}_R) \mathbin{/\!/} (1, v) \succ^* s_0 \qquad \textit{(sound)}$$
$$(1, \mathtt{pcp\_bsm}_R) \mathbin{/\!/} (1, v) \rightsquigarrow s \to \text{iBPCP}\ R \wedge s = s_0 \qquad \textit{(complete)}$$

where $s_0 = (0, v_0)$ *and* $v_0 = [s, u, l, a := []]$.

Let us remark that the length of $\mathtt{pcp\_bsm}_R$ is $86 + 3|R| + \mathtt{size}\,R$ where $\mathtt{size}\,R$ is the length of all concatenated upper and lower traces in R. This allows us to conclude:

Theorem 5.7. iBPCP $\preceq$ BSM.

6 A Certified Low-Level Compiler

We present the implementation of a certified compiler for low-level languages as defined in Section 4, to be used in later Section 7.3 to reduce BSM-termination to MM-termination.

Let us start with a description of the problem. We are given two models of computation X and Y as presented in Section 4, each with their own set of instructions and their own semantics described by step relations. We adapt the notation of Section 4 to discuss about those two different models at the same time by decorating the notations with either X or Y, hence $\mathcal{P} \;/\!\!/_X\; s_1 \succ^3 s_2$ means program $\mathcal{P}$ of model X can compute state s_2 in 3 steps when starting from state s_1, while $\mathcal{Q} \;/\!\!/_Y\; s\downarrow$ means that the computation of program $\mathcal{Q}$ of model Y terminates when starting from state s. The model X will be the source while Y will be the target model of the compiler.

The compiler consists of transforming any program $\mathcal{P}$ in model X into a program $\mathcal{Q}$ in model Y such that $\mathcal{Q}$ simulates the behaviour of $\mathcal{P}$ faithfully. We model the correspondence between the configurations of machines in model X and in model Y by a *simulation* binary relation $\bowtie : \mathbb{C}_X \to \mathbb{C}_Y \to \mathbb{P}$. A typical soundness result would be

$$\begin{aligned} &v_1 \bowtie w_1 \wedge \mathcal{P} \;/\!\!/_X\; (i_1, v_1) \succ^* (i_2, v_2) \\ \to\; &\exists (j_2, w_2),\, v_2 \bowtie w_2 \wedge \mathcal{Q} \;/\!\!/_Y\; (j_1, w_1) \succ^* (j_2, w_2) \end{aligned}$$

meaning that this particular computation of $\mathcal{P}$ from (i_1, v_1) is simulated by some computation of $\mathcal{Q}$ from (j_1, w_1) preserving the simulation relation between the configurations in X and Y. However, such a result only expresses a relation between configurations, not between PC values. The relation between PC values is going to be made more precise using a linker.

As it is generic, our compiler cannot cope with the particular behaviour of individual instructions because that really depends on the choice of X and Y: for instance, there cannot exist a correct compiler if model X is more expressive than model Y. As a result, the generic compiler assumes as input a *one-to-many instruction compiler* that transforms a single instruction in $\mathbb{I}_X$ into a list of instructions in $\mathbb{L}\,\mathbb{I}_Y$. The problem solved by our generic compiler is transforming this individual instruction compiler into a program compiler while preserving soundness.

Because the replacement of X instructions by Y instructions is one-to-many, absolute jumps must be relocated at addresses that depend on the whole program, not only on the instruction that is being compiled. For instance, consider the absolute jumps p and q embedded in the BSM-instruction $\texttt{POP}\ \alpha\ p\ q$ which is compiled in 16 MM-instructions in Section 7.3. These PC addresses p/q must be recomputed before compiling the $\texttt{POP}\ \alpha\ p\ q$ instruction. But their new values depend on the content of the compiled program *ahead of the instructions they point at*. If they point forward, a chicken-and-egg problem appears. To solve it, traditional compilers replace p/q with symbolic addresses and the whole program is compiled in a symbolic address space of graph-like structure. Symbols are resolved by a *linker* in a later phase that maps the graph of instructions into a sequence again.

Here, we can avoid this quite complex machinery by assuming a specific feature of our instruction compiler: even though the list $ll_\rho : \mathbb{L}\,\mathbb{I}_Y$ of Y-instructions corresponding to some X-instruction $\rho : \mathbb{I}_X$ might depend on where ρ is located in the compiled code (and also on where the absolute and relative[3] jumps which ρ contains are remapped), the *length* $|ll_\rho|$ *does not depend on those actual positions*. So there is a first abstract compilation phase that only computes the length of codes, from which we can deduce a linker that is going to be compatible with a later concrete compilation phase that computes actual instructions at their definitive location as given by the linker.

Let us now describe the compiler more formally. We assume an instruction compiler and a code length function:

$$\begin{aligned} &\mathsf{icomp} : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{I}_X \to \mathbb{L}\,\mathbb{I}_Y \\ &\mathsf{ilen} \quad\; : \mathbb{I}_X \to \mathbb{N} \end{aligned}$$

where $\mathsf{icomp}\ lnk\ i\ \rho$ compiles instruction ρ into $ll_\rho : \mathbb{L}\,\mathbb{I}_Y$ at source position i using linker lnk to remap absolute and relative jumps, and the length of ll_ρ is given by $\mathsf{ilen}\ \rho$. Hence, we assume the hypothesis

$$H_{\mathsf{ilen}} : \quad \forall\, lnk\, i\, \rho,\ |\mathsf{icomp}\ lnk\ i\ \rho| = \mathsf{ilen}\ \rho$$

so that the length of compiled instructions only depends on instruction ρ, not where it is positioned (i.e. i) or where jumps are remapped (i.e. lnk). In contrast, the actual code $\mathsf{icomp}\ lnk\ i\ \rho$ almost always depends on lnk and i.

Let us now come to the semantic soundness hypothesis on icomp. For this, a step relation for X and a step relation for Y must be given. For simplicity of arguments, we assume that the step relation for X is total and that the step relation for Y is deterministic (i.e. functional).

$$\begin{aligned} &H_X : \text{the step relation } \succ_X \text{ is total} \\ &H_Y : \text{the step relation } \succ_Y \text{ is deterministic} \end{aligned}$$

We do not view totality as a strong requirement. It could be noted however that determinism would not be satisfied in case the target code involves some form of parallelism.

Given a simulation relation $\bowtie : \mathbb{C}_X \to \mathbb{C}_Y \to \mathbb{P}$ between the configurations of X and Y, the critical soundness assumption we make on icomp is the following:

$$\begin{aligned} H_{\mathsf{icomp}} : \forall\, &lnk\, \rho\, (i_1, v_1)\, (i_2, v_2)\ w_1, \\ &\rho \;/\!\!/_X\; (i_1, v_1) \succ (i_2, v_2) \\ &\wedge\ v_1 \bowtie w_1 \wedge lnk(1 + i_1) = \mathsf{ilen}\ \rho + lnk(i_1) \\ \to\; &\exists w_2,\, (lnk\ i_1, ll_\rho) \;/\!\!/_Y\; (lnk\ i_1, w_1) \succ^+ (lnk\ i_2, w_2) \\ &\wedge\ v_2 \bowtie w_2 \end{aligned}$$

[3] Although instructions might not refer to specific relative addresses explicitly, the default incrementation of the PC by one makes +1 an almost ubiquitous relative address.

where ll_ρ is shorthand for $ll_\rho := \mathsf{icomp}\ lnk\ i_1\ \rho$. It requires some explanations: H_{icomp} states that if ll_ρ is the result of the compilation of instruction ρ at source position i_1 using linker lnk, if the X-step relation for ρ transforms state (i_1, v_1) into state (i_2, v_2) and if w_1 simulates v_1, then the compiled code ll_ρ at its target address as given by lnk computes some state $(lnk\ i_2, w_2)$ from state $(lnk\ i_1, w_1)$ in the Y semantics and w_2 simulates v_2. The simulated computation must make progress, hence ll_ρ can never be empty. Additionally, the pre-condition $lnk(1 + i_1) = \mathsf{ilen}\ \rho + lnk(i_1)$ assumes that the linker lnk preserves the +1 (implicit) relative address.

These assumptions, in particular H_{icomp} which is the most important, are sufficient to build our program compiler and show its correctness, of which the proof is decomposed in two phases: a semantic one and a syntactic one.

6.1 Semantic Correctness of Compiled Code

Under the previous assumptions, suppose that we are given a source X-program $\mathcal{P}$, a linker lnk and a compiled Y-program $\mathcal{Q}$. We give a syntactic correctness criterion which is sufficient to establish that $\mathcal{Q}$ correctly simulates $\mathcal{P}$. The criterion

$$H_{\mathcal{P}\mathcal{Q}} : \forall\, i\ \rho,\ (i, [\rho]) <_{\mathsf{sc}} \mathcal{P} \to \wedge \begin{cases} (lnk\ i, \mathsf{icomp}\ lnk\ i\ \rho) <_{\mathsf{sc}} \mathcal{Q} \\ lnk(1 + i) = \mathsf{ilen}\ \rho + lnk(i) \end{cases}$$

means that whenever instruction ρ occurs in $\mathcal{P}$ at position i then its corresponding compiled code $\mathsf{icomp}\ lnk\ i\ \rho$ occurs at position $lnk\ i$ in $\mathcal{Q}$ and lnk respects the $1 + i$ relative address. This condition does not involve the semantics chosen for X and Y at all, it is a purely syntactic assumption which, in combination with H_{icomp}, is sufficient to ensure that $\mathcal{Q}$ correctly simulates $\mathcal{P}$.

Theorem 6.1. *Under assumptions $H_{\mathsf{icomp}}, H_{\mathsf{ilen}}, H_X, H_Y$ and $H_{\mathcal{P}\mathcal{Q}}$, the program $\mathcal{Q}$ soundly simulates $\mathcal{P}$, that is for any (i_1, v_1), (i_2, v_2) and w_1, the following property holds:*

$$v_1 \bowtie w_1 \wedge \mathcal{P} \mathbin{/\!\!/_X} (i_1, v_1) \succ^* (i_2, v_2)$$
$$\to \exists w_2,\ v_2 \bowtie w_2 \wedge \mathcal{Q} \mathbin{/\!\!/_Y} (lnk\ i_1, w_1) \succ^* (lnk\ i_2, w_2)$$

Proof. By induction on the computation of $\mathcal{P}$. □

The completeness of $\mathcal{Q}$ is more complicated to express and prove because *not every computation of $\mathcal{Q}$ corresponds to a computation of $\mathcal{P}$*. To ensure that property, we must restrict the computations of $\mathcal{Q}$ to those which start at some $lnk\ i$ (obviously) but we must also assume termination using big-step semantics.

Theorem 6.2. *Under assumptions $H_{\mathsf{icomp}}, H_{\mathsf{ilen}}, H_X, H_Y$ and $H_{\mathcal{P}\mathcal{Q}}$, the program $\mathcal{Q}$ completely simulates $\mathcal{P}$, that is for any (i_1, v_1) and w_1, the following property holds:*

$$v_1 \bowtie w_1 \wedge \mathcal{Q} \mathbin{/\!\!/_Y} (lnk\ i_1, w_1)\!\downarrow \to \mathcal{P} \mathbin{/\!\!/_X} (i_1, v_1)\!\downarrow$$

Proof. By complete induction on the length q of the terminating computation $\mathcal{Q} \mathbin{/\!\!/} (lnk\ i_1, w_1) \rightsquigarrow^q s$. □

Hence, to get a correct (sound and complete) compiler from a sound instruction compiler, it is enough to produce code that satisfies $H_{\mathcal{P}\mathcal{Q}}$ which is a purely syntactic criterion.

6.2 A Syntactically Correct Compiler

Obtaining lnk and $\mathcal{Q}$ satisfying condition $H_{\mathcal{P}\mathcal{Q}}$ from $\mathcal{P}$ is not completely trivial because as said earlier, the linker lnk which represents the remapping of the address space of the source program $\mathcal{P}$, depends on the length of compiled code (hence $\mathcal{Q}$) and $\mathcal{P}$ needs lnk to be compiled correctly. The circularity is broken under the H_{ilen} assumption that isolates code length from target address space considerations.

More formally, given source code $\mathcal{P} = (i, [\rho_0; \ldots; \rho_{n-1}])$ and a destination address j where the compiled code is supposed to start, we first build lnk such that

$$\mathsf{lnk}(k + i) = j + \mathsf{ilen}\ \rho_0 + \cdots + \mathsf{ilen}\ \rho_{k-1} \quad \text{for } k \in [0, n]$$

and then the compiled code $\mathcal{Q}$ as

$$\mathcal{Q} := (j, \mathsf{icomp}\ \mathsf{lnk}\ i\ \rho_0 +\!\!+ \cdots +\!\!+ \mathsf{icomp}\ \mathsf{lnk}\ (n - 1 + i)\ \rho_{n-1}).$$

There are some subtleties of secondary importance related to where source PC values outside of $\mathcal{P}$ should be remapped.[4] For our purpose, we simply map them all to $\mathsf{lnk}(n + i)$. This simple linker/compiler satisfies the syntactic correctness criterion $H_{\mathcal{P}\mathcal{Q}}$ and we deduce the following correction theorem.

Theorem 6.3 (Compiler). *Assuming $H_{\mathsf{icomp}}, H_{\mathsf{ilen}}, H_X, H_Y$, for any X-program $\mathcal{P}$ and any destination address j, one can compute a linker $\mathsf{lnk} : \mathbb{N} \to \mathbb{N}$ and a Y-program $\mathcal{Q}$ with the following properties:*

1. $j = \mathsf{start}\ \mathcal{Q} = \mathsf{lnk}(\mathsf{start}\ \mathcal{P})$;
2. *for any i,* $\mathsf{out}\ i\ \mathcal{P} \to \mathsf{lnk}\ i = \mathsf{end}\ \mathcal{Q}$;
3. *the following holds for any i_1, v_1, i_2, v_2, w_1:*

$$v_1 \bowtie w_1 \wedge \mathcal{P} \mathbin{/\!\!/_X} (i_1, v_1) \rightsquigarrow (i_2, v_2)$$
$$\to \exists w_2,\ v_2 \bowtie w_2 \wedge \mathcal{Q} \mathbin{/\!\!/_Y} (\mathsf{lnk}\ i_1, w_1) \rightsquigarrow (\mathsf{lnk}\ i_2, w_2)$$

4. *the following holds for any i_1, v_1, w_1, j_2, w_2:*

$$v_1 \bowtie w_1 \wedge \mathcal{Q} \mathbin{/\!\!/_Y} (\mathsf{lnk}\ i_1, w_1) \rightsquigarrow (j_2, w_2)$$
$$\to \exists\, i_2\ v_2,\ v_2 \bowtie w_2 \wedge \mathcal{P} \mathbin{/\!\!/_X} (i_1, v_1) \rightsquigarrow (i_2, v_2) \wedge j_2 = \mathsf{lnk}\ i_2.$$

This concludes our implementation of a generic and certified compiler, as a functor that maps a sound instruction compiler into a correct program compiler.

6.3 Discussion

Our generic compiler is certainly not the only certified compiler around, CakeML [22] (ML compiler to assembly, certified in HOL) and CompCert [27] (C compiler to assembly, certified in Coq) being among the most famous ones. We make some comparisons with two Coq-certified compilers [5, 27]. Unfortunately for our purpose, these two compilers both involve specific source and target languages which are of little

[4] Depending of the intended application, it could be useful to distinguish the remappings of source PC addresses outside of $\mathcal{P}$.

use for the BSM-MM reduction we need. These compilers are more ambitious than our intended BSM-MM compiler: source and target languages are more complex both syntactically and semantically. In the case of CompCert [27], there is also an aim at certifying most of the compilation tool-chain, up to near state-of-the-art optimisations.

Hence, we do not claim to have solved a very difficult open problem with our generic compiler. Nonetheless, we want to point out two of its main specificities. First, we have identified a small set of assumptions that suffice at proving correctness of the produced compiled code. We hope that these could serve as guidelines for other developments to come. More critically in our setting which consists in building reductions between undecidable problems, our compiler comes not only with a soundness theorem 6.1 comparable to those of [5, 27], but it also comes with a proof of *completeness* (Theorem 6.2) stating that termination of compiled code implies termination of source code.

In the cases of [5] or CompCert [27], this might be a non-issue since the intended use is only for terminating programs. To the best of our understanding, this question of completeness does not seem to have been considered by any of these projects. However, to establish undecidability results — all the more in a constructive setting, — it is not possible to assume termination and thus we really need a completeness result to show the correctness of compiler-based reductions.

7 Minsky Machines

Minsky Machines (MM) [31] have n registers with natural number values. Similar to BSM configurations, we define MM configurations as vectors of registers values, i.e. $\mathbb{C} := \mathbb{N}^n$. Instructions are defined by

$$\mathbb{I}_n ::= \mathtt{INC}\ (\alpha : \mathbb{F}_n) \mid \mathtt{DEC}\ (\alpha : \mathbb{F}_n)\ (p : \mathbb{N}).$$

7.1 Semantics for MM

The rules defining the step relation for MM instructions are:

$$\begin{aligned}
&\mathtt{INC}\ \alpha \mathbin{/\!/} (i, v) \succ (1 + i, v[\alpha := 1 + n]) && \text{if } v[\alpha] = n\\
&\mathtt{DEC}\ \alpha\ p \mathbin{/\!/} (i, v) \succ (p, v) && \text{if } v[\alpha] = 0\\
&\mathtt{DEC}\ \alpha\ p \mathbin{/\!/} (i, v) \succ (1 + i, v[\alpha := n]) && \text{if } v[\alpha] = 1 + n
\end{aligned}$$

That means, $\mathtt{INC}\ \alpha$ increases the value of register α by one. $\mathtt{DEC}\ \alpha\ p$ decreases the value of register α by one if that is possible and increases the PC, or, if the register is already 0, jumps to PC value p. As is the case for BSM, the step relation for MM is both deterministic and total.

Finally, we use a more restricted halting problem for MM than for BSM, where we start execution with PC at 1 and expect the machine to halt with PC at 0 and on the configuration where all registers are null:

$$\mathsf{MM}\big(n : \mathbb{N}, P : \mathbb{L}\,\mathbb{I}_n, v : \mathbb{N}^n\big) := (1, P) \mathbin{/\!/} (1, v) \rightsquigarrow (0, \vec{0}).$$

This restricted halting problem is well-suited for a reduction to linear logic. We could use a more general halting problem and then reduce this more general problem to the special problem. However, this would require a dedicated transformation on MM programs, which we circumvent by reducing BSM to the special version directly.

7.2 Simulating Binary Stacks with Numbers

The reduction from BSM to MM is derived as a direct consequence of our implementation of a generic certified low-level compiler. Basically to get a correct compiler from BSM to MM, the only remaining tasks are: (a) simulate binary stacks with natural numbers, and (b) simulate individual BSM-instructions with lists of MM-instructions. We give a high-level overview of the principal ideas and the main soundness and completeness results.

BSM programs manipulate binary stacks in $\mathbb{L}\,\mathbb{B}$ while MM programs manipulate natural numbers in $\mathbb{N}$. Hence to represent $\mathbb{L}\,\mathbb{B}$ with $\mathbb{N}$, we implement multiplication and division by 2. For instance, we show the following fact.

Fact 7.1. *For any $n : \mathbb{N}$, $x \neq q \neq r : \mathbb{F}_n$ and $i : \mathbb{N}$, the list* `mm_div2` *of 6 MM-instructions*

$$[\mathtt{DEC}\ x\ (6+i); \mathtt{INC}\ r; \mathtt{DEC}\ x\ (6+i); \mathtt{DEC}\ r\ (4+i); \mathtt{INC}\ q; \mathtt{DEC}\ r\ i]$$

implements division by 2 because it satisfies the following specification for all k, b, v:

$$\begin{aligned}
&v[q] = 0 \land v[r] = 0 \land v[x] = b + 2.k\\
\to\ &(i, \mathtt{mm_div2}) \mathbin{/\!/} (i, v) \succ^+ (6 + i, v[x := 0, q := k, r := b])
\end{aligned}$$

Note that $b : \mathbb{B}$ is cast into $0/1 : \mathbb{N}$ to simplify the above statement. We also need transfer (resp. nullify) programs which adds the value of register x to y, nullifying x at the same time (resp. nullifies one or several registers). These programs are (somewhat) obvious to write and prove correct.

We can then encode a binary stack $l : \mathbb{L}\,\mathbb{B}$ as a natural number $\mathsf{s2n}\,l : \mathbb{N}$ using:

$$\mathsf{s2n}\,[] := 1 \quad \mathsf{s2n}\,(0 :: l) := 2 \cdot \mathsf{s2n}\,l \quad \mathsf{s2n}\,(1 :: l) := 1 + 2 \cdot \mathsf{s2n}\,l$$

and simulate $\mathtt{PUSH}\ x\ 0$, $\mathtt{PUSH}\ x\ 1$ and $\mathtt{POP}\ x\ j\ e$ instructions.

Lemma 7.2. *For any $n : \mathbb{N}$, $x \neq z \neq t : \mathbb{F}_n$ and $i : \mathbb{N}$, there is a list* `mm_push_0` *of 7 instructions which satisfies the following specification for all l, v:*

$$\begin{aligned}
&v[t] = 0 \land v[z] = 0 \land v[x] = \mathsf{s2n}\,l\\
\to\ &(i, \mathtt{mm_push_0}) \mathbin{/\!/} (i, v) \succ^+ (7 + i, v[x := \mathsf{s2n}(0 :: l)])
\end{aligned}$$

Lemma 7.3. *For any $n : \mathbb{N}$, $x \neq z \neq t : \mathbb{F}_n$ and $i : \mathbb{N}$, there is a list* `mm_push_1` *of 8 instructions which satisfies the following specification for all l, v:*

$$\begin{aligned}
&v[t] = 0 \land v[z] = 0 \land v[x] = \mathsf{s2n}\,l\\
\to\ &(i, \mathtt{mm_push_1}) \mathbin{/\!/} (i, v) \succ^+ (8 + i, v[x := \mathsf{s2n}(1 :: l)])
\end{aligned}$$

Lemma 7.4. *For any $n : \mathbb{N}$, $x \neq z \neq t : \mathbb{F}_n$ and $i, j, k, e : \mathbb{N}$, there is a list* `mm_pop` *of 16 instructions which satisfies the*

following specifications for any l and v s.t. $v[t] = v[z] = 0$:

$$v[x] = \mathsf{s2n}\,[] \longrightarrow (i, \mathtt{mm\_pop}) \;/\!/\; (i, v) >^+ (e, v)$$
$$v[x] = \mathsf{s2n}(0 :: l) \longrightarrow (i, \mathtt{mm\_pop}) \;/\!/\; (i, v) >^+ (j, v[x := \mathsf{s2n}\, l])$$
$$v[x] = \mathsf{s2n}(1 :: l) \longrightarrow (i, \mathtt{mm\_pop}) \;/\!/\; (i, v) >^+ (k, v[x := \mathsf{s2n}\, l])$$

Hence, we simulate any individual BSM instruction by a (small) list of MM instructions that operate on stacks encoded using s2n. Using Theorem 6.3 of Section 6, we obtain a correct compiler for whole BSM programs to MM programs.

7.3 BSM Reduces to MM

We describe the correctness and completeness results we obtain from the generic compiler of Theorem 6.3. Assume that we are given a BSM program (i, P) with n binary stack registers, we compile (i, P) at address 1 into a linker $\mathsf{lnk} : \mathbb{N} \longrightarrow \mathbb{N}$ and a MM program Q starting at address 1 with $2 + n$ registers. Each stack register $p : \mathbb{F}_n$ is s2n encoded into the register $2 + p : \mathbb{F}_{2+n}$ of Q. The registers $0, 1 : \mathbb{F}_{2+n}$ of Q serve as spare registers for internal computations. The simulation invariant

$$v \bowtie w := w[0] = 0 \wedge w[1] = 0 \wedge \forall p, w[2 + p] = \mathsf{s2n}(v[p])$$

ensures the preservation of the encoding during execution. lnk satisfies $\mathsf{lnk}\, i = 1$ and for any j for which out j (i, P) holds, we have $\mathsf{lnk}\, j = \mathsf{end}(1, Q)$. Moreover, the code $(1, Q)$ satisfies the following soundness and completeness results:

Fact 7.5. *For any i_1, i_2, v_1, v_2, w_1, the following holds:*

$$v_1 \bowtie w_1 \wedge (i, P) \;/\!/\; (i_1, v_1) \rightsquigarrow (i_2, v_2)$$
$$\longrightarrow \exists w_2, v_2 \bowtie w_2 \wedge (1, Q) \;/\!/\; (\mathsf{lnk}\, i_1, w_1) \rightsquigarrow (\mathsf{lnk}\, i_2, w_2).$$

Fact 7.6. *For any i_1, v_1, w_1, j_2, w_2, the following holds:*

$$v_1 \bowtie w_1 \wedge (1, Q) \;/\!/_Y\; (\mathsf{lnk}\, i_1, w_1) \rightsquigarrow (j_2, w_2)$$
$$\longrightarrow \exists\, i_2\, v_2, v_2 \bowtie w_2 \wedge (i, P) \;/\!/_X\; (i_1, v_1) \rightsquigarrow (i_2, v_2).$$

From those correctness results, it is easy to derive a simulator for BSM termination as MM termination.

Theorem 7.7 (BSM simulator)**.** *Let (i, P) be a BSM-program with n binary stacks. One can compute a $2 + n$ registers MM-program* bsm_mm *with the following specification. For any v, with $w := [\mathsf{s2n}\, l \mid l \in v]$, we have:*

$$(i, P) \;/\!/\; (i, v)\downarrow \leftrightarrow (1, \mathtt{bsm\_mm}) \;/\!/\; (1, 0 :: 0 :: w) \rightsquigarrow (0, \vec{0}).$$

Proof. The code of bsm_mm consists of the compiled code $(1, Q)$ of (i, P) followed by some obvious code nullifying all the variables after the execution of $(1, Q)$ is terminated and then jumping to PC value 0. The nullifying code uses the fact that any output state (j_2, w_2) of $(1, Q)$ satisfies $j_2 = \mathsf{end}(1, Q)$ and $w_2[0] = 0$ to nullify the other registers using register 0 as empty register. We critically need both the soundness Fact 7.5 and completeness Fact 7.6 to get the above equivalence. □

This allows us to conclude:

Theorem 7.8. BSM $\preceq$ MM.

8 Intuitionistic Linear Logic

$$\frac{}{A \vdash A} \qquad \frac{\Gamma \vdash A \qquad \Delta, B \vdash C}{\Gamma, \Delta, A \multimap B \vdash C} \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B}$$

$$\frac{\Gamma, A \vdash C}{\Gamma, A \,\&\, B \vdash C} \qquad \frac{\Gamma, B \vdash C}{\Gamma, A \,\&\, B \vdash C} \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \,\&\, B}$$

$$\frac{\Gamma, A \vdash B}{\Gamma, !\,A \vdash B} \qquad \frac{\Gamma \vdash B}{\Gamma, !\,A \vdash B} \qquad \frac{\Gamma, !\,A, !\,A \vdash B}{\Gamma, !\,A \vdash B} \qquad \frac{!\,\Gamma \vdash B}{!\,\Gamma \vdash !\,B}$$

Figure 1. The $\mathsf{S}_{\mathsf{ILL}}$ sequent calculus.

The reduction of MM to entailment in *Intuitionistic Linear Logic (ILL)* [15] implements the method presented in [25] and fully described in [26]. The undecidability of (I)LL was first established in [29] and a rich literature followed [19, 20, 28], but [25] singled out the elementary fragment of ILL (eILL) together with a short proof of undecidability by reducing the termination of two counters Minsky machines.

Here we present the reduction of termination of arbitrary MM (with n counters) and we verify it in Coq. The proof consists first of the reduction of MM to entailment in eILL, and then the proof of the (faithful) embedding of eILL into ILL. The latter proof actually takes place in the $\{\&, \multimap, !\}$ cut-free fragment of ILL, i.e. we do not need the $\{\otimes, \oplus, \bot, \top, 1\}$ logical connectives nor the cut rule to build the embedding.

As a way to limit the technical material we need to present, we only introduce the $\{\&, \multimap, !\}$ fragment in this paper. However, the Coq code contains the proof of the embedding of eILL into the whole ILL.[5] Hence from now on, we will abusively denote ILL for its restriction to the $\{\&, \multimap, !\}$ fragment.

The syntax of (the $\{\&, \multimap, !\}$ fragment of) ILL is described by the BNF-grammar $A ::= v \mid A \,\&\, A \mid A \multimap A \mid !\,A$ where v belongs to a *countably infinite* set of logical variables. A *sequent* is a pair denoted $\Gamma \vdash A$ composed of a multiset Γ of formulæ and a single formula A on the right of the $\vdash$ sign. The *sequent calculus* $\mathsf{S}_{\mathsf{ILL}}$ is described in Fig. 1. In the last rule, $!\,\Gamma$ denotes $!\,\Gamma := !\,A_1, \ldots, !\,A_n$ when $\Gamma = A_1, \ldots, A_n$. Using $\mathsf{S}_{\mathsf{ILL}}$, we define the entailment relation: the hypotheses in Γ entail A when $\Gamma \vdash A$ has a derivation in $\mathsf{S}_{\mathsf{ILL}}$.

8.1 eILL Reduces to ILL

ILL contains a sub-fragment called *elementary Intuitionistic Linear Logic (eILL)* first spotted in [25]. It is not a syntactic fragment at the level of formulæ, but it is a syntactic fragment at the level of sequents. A sequent in eILL is a sequent of ILL of the form $!\,\Sigma, \Gamma \vdash u$, where Γ is a multiset composed of logical variables only and u is a unique logical variable. On the contrary, Σ is composed of formulæ called *command*

[5] This means that our method proves the embedding of eILL into both the $\{\&, \multimap, !\}$ fragment and the whole ILL. However, we do not show or formalise the embedding of the fragment into the whole: it involves a proof of either cut-elimination, or a larger phase semantics development.

formulæ [26] of the form $(u \multimap v) \multimap w$, $u \multimap (v \multimap w)$ and $(u \,\&\, v) \multimap w$ where u, v and w are logical variables. So eILL sequents have a very simple/flat syntactic structure.

$$\frac{}{!\,\Sigma, u \vdash u}\ \text{Ax} \qquad \frac{!\,\Sigma, \Gamma, u \vdash v}{!\,\Sigma, \Gamma \vdash w}\ (u \multimap v) \multimap w \in \Sigma$$

$$\frac{!\,\Sigma, \Gamma \vdash u \qquad !\,\Sigma, \Delta \vdash v}{!\,\Sigma, \Gamma, \Delta \vdash w}\ u \multimap (v \multimap w) \in \Sigma$$

$$\frac{!\,\Sigma, \Gamma \vdash u \qquad !\,\Sigma, \Gamma \vdash v}{!\,\Sigma, \Gamma \vdash w}\ (u \,\&\, v) \multimap w \in \Sigma$$

Figure 2. The $\mathrm{G_{eILL}}$ goal-directed sequent calculus.

Although entailment in eILL could be described by $\mathrm{S_{ILL}}$, for the purpose of reducing MM, it is more conveniently described by the goal directed rules of $\mathrm{G_{eILL}}$ in Fig. 2. The reason is that they directly encode MM small-step semantics as shall be explained in Section 8.2. Notice however that while sequents of eILL are stable under (backwards application of) the rules of $\mathrm{G_{eILL}}$, they are not stable under the rules of $\mathrm{S_{ILL}}$, which means that proof-search from an eILL sequent in $\mathrm{S_{ILL}}$ might produce sequents which do not belong to eILL.

Given a commutative monoid $(M, \bullet, \epsilon)$, for any $X, Y \subseteq M$, we define the *point-wise extension* by $X \bullet Y := \{x \bullet y \mid x \in X \wedge y \in Y\}$ and its *linear adjunct* as $X \multimap Y := \{k \in M \mid \{k\} \bullet X \subseteq Y\}$ providing a residuated monoidal structure on the powerset $M \to \mathbb{P}$. Given any semantic interpretation $[\![\cdot]\!] \subseteq M$ of logical variables, we extend it inductively to ILL via *trivial phase semantics*:

$$[\![A \,\&\, B]\!] := [\![A]\!] \cap [\![B]\!] \qquad [\![!\,A]\!] := \{\epsilon\} \cap [\![A]\!]$$
$$[\![A \multimap B]\!] := [\![A]\!] \multimap [\![B]\!]$$

and to multisets by $[\![A_1, \ldots, A_n]\!] := [\![A_1]\!] \bullet \cdots \bullet [\![A_n]\!]$. An ILL sequent $\Gamma \vdash A$ is *valid in that interpretation* if $[\![\Gamma]\!] \subseteq [\![A]\!]$. A sequent is *valid in a commutative monoid* $(M, \bullet, \epsilon)$ if it is valid for any trivial interpretation $[\![\cdot]\!] \subseteq M$ in that monoid.

Theorem 8.1. *The three following results hold:*

1. *If derivable in* $\mathrm{G_{eILL}}$, $!\,\Sigma, \Gamma \vdash u$ *is also derivable in* $\mathrm{S_{ILL}}$*;*
2. *An ILL sequent derivable in* $\mathrm{S_{ILL}}$ *is valid in every commutative monoid;*
3. *Let* n *be greater than the number of logical variables of* Σ, Γ*. If* $!\,\Sigma, \Gamma \vdash u$ *is valid in the free commutative monoid* $(\mathbb{N}^n, +, \vec{0})$ *then* $!\,\Sigma, \Gamma \vdash u$ *is derivable in* $\mathrm{G_{eILL}}$*.*

Notice that trivial phase semantics is also sound for the whole ILL, not just the $(\&, \multimap, !)$-fragment ILL or eILL. But it is only complete for eILL, not for the $(\&, \multimap, !)$-fragment nor for the whole ILL because the interpretation of the bang ! is too restricted compared to general phase semantics [26].

As is conventional in this paper, we also denote by eILL (resp. ILL) the entailment problem in eILL (resp. ILL): for a given sequent of eILL (resp. ILL), does it have a derivation in $\mathrm{G_{eILL}}$ (resp. $\mathrm{S_{ILL}}$)? Then we can show the following reduction:

Theorem 8.2. $\mathsf{eILL} \preceq \mathsf{ILL}$.

Proof. The reduction is just the identity map: a sequent of eILL has a derivation in $\mathrm{G_{eILL}}$ if and only if it has one in $\mathrm{S_{ILL}}$ and this is a direct consequence of Theorem 8.1. □

8.2 MM Reduces to eILL

We show how to faithfully encode MM-termination by entailment in $\mathrm{G_{eILL}}$. For this, we describe how to transform a n registers MM-program P and a vector $v : \mathbb{N}^n$ into an eILL-sequent, s.t. the $\mathrm{G_{eILL}}$-entailment of the sequent simulates the predicate $(1, P) \mathbin{/\!/} (1, v) \rightsquigarrow (0, \vec{0})$.

We fix n and $P = [\rho_1; \ldots; \rho_l]$. Let us partition the countably infinite set of logical variables into disjoint sets

$$\{x_0, \ldots, x_{n-1}\} \uplus \{\overline{x}_0, \ldots, \overline{x}_{n-1}\} \uplus \{q_0, q_1, \ldots\}.$$

Let us describe the eILL-sequent $!\,\Sigma, \Gamma_v \vdash q_1$ simulating $(1, P) \mathbin{/\!/} (1, v) \rightsquigarrow (0, \vec{0})$. We start with the formulæ in Σ. For every labelled instruction $i : \rho_i$ of $(1, P)$, we add one or two command formulæ in Σ according to:

$$\begin{array}{rcl} i : \mathtt{INC}\ p & \rightsquigarrow & (x_p \multimap q_{i+1}) \multimap q_i \\ i : \mathtt{DEC}\ p\ j & \rightsquigarrow & x_p \multimap (q_{i+1} \multimap q_i) \text{ and } (\overline{x}_p \,\&\, q_j) \multimap q_i. \end{array}$$

In addition, we add the following $1 + n^2$ command formulæ to Σ independently of P. These just depend on n:

$$\{(q_0 \multimap q_0) \multimap q_0\} \uplus \{(\overline{x}_i \multimap \overline{x}_i) \multimap \overline{x}_i \mid i < n\} \uplus \{x_j \multimap (\overline{x}_i \multimap \overline{x}_i) \mid i \neq j < n\}.$$

We define the multiset $\Gamma_v := \{v[0].x_0, \ldots, v[n-1].x_{n-1}\}$ where $p.x_i$ means repeating p times x_i in the multiset. We show the following reduction.

Lemma 8.3. *We have* $(1, P) \mathbin{/\!/} (i, v) \rightsquigarrow (0, \vec{0})$ *if and only if the sequent* $!\,\Sigma, \Gamma_v \vdash q_i$ *is provable in* $\mathrm{G_{eILL}}$*.*

Proof. We show the *only if part* by structural induction on the predicate $(1, P) \mathbin{/\!/} (i, v) \succ^* (0, \vec{0})$. In the base case we have $i = 0$ and $v = \vec{0}$ and we check $!\,\Sigma, \emptyset \vdash q_0$:

$$\dfrac{\dfrac{}{!\,\Sigma, q_0 \vdash q_0}\ \text{Ax}}{!\,\Sigma, \emptyset \vdash q_0}\ (q_0 \multimap q_0) \multimap q_0 \in \Sigma$$

If the last rule is

$$\frac{\mathtt{INC}\ p \mathbin{/\!/} (i, v) \succ s \qquad (1, P) \mathbin{/\!/} s \succ^* (0, \vec{0})}{(1, P) \mathbin{/\!/} (i, v) \succ^* (0, \vec{0})}$$

then $s = (i + 1, v + 1.x_p)$ and we build the following proof

$$\dfrac{\dfrac{\cdots}{!\,\Sigma, \Gamma_v, x_p \vdash q_{i+1}}}{!\,\Sigma, \Gamma_v \vdash q_i}\ (x_p \multimap q_{i+1}) \multimap q_i \in \Sigma$$

where the top dots symbolize a sub-proof corresponding to the induction hypothesis. Notice that $(x_p \multimap q_{i+1}) \multimap q_i$ was explicitly added to Σ to encode instruction $i : \mathtt{INC}\ p$.

We do the same for $i : \mathtt{DEC}\ p\ j$. If the last rule is

$$\frac{\mathtt{DEC}\ p\ j \mathbin{/\!/} (i, v) \succ s \qquad (1, P) \mathbin{/\!/} s \succ^* (0, \vec{0})}{(1, P) \mathbin{/\!/} (i, v) \succ^* (0, \vec{0})}$$

then there are two cases depending on whether the value of v at coordinate p is 0 or not. If $v[p] \neq 0$ then $v = w + 1.x_p$, $\Gamma_v = \Gamma_w, x_p$ and $s = (i + 1, w)$ and we build the following proof using the induction hypothesis for the top-right dots:

$$\frac{\dfrac{}{!\,\Sigma, x_p \vdash x_p}\ \mathrm{Ax} \qquad \dfrac{\cdots}{!\,\Sigma, \Gamma_w \vdash q_{i+1}}}{!\,\Sigma, \Gamma_w, x_p \vdash q_i}\ x_p \multimap (q_{i+1} \multimap q_i) \in \Sigma$$

The case where $v[p] = 0$ is a bit more complicated. In that case, $x_p \notin \Gamma_v$ and we have $s = (j, v)$. We build the proof as:

$$\frac{\dfrac{\cdots}{!\,\Sigma, \Gamma_v \vdash \overline{x}_p}\ x_p \notin \Gamma_v \qquad \dfrac{\cdots}{!\,\Sigma, \Gamma_v \vdash q_j}}{!\,\Sigma, \Gamma_v \vdash q_i}\ (\overline{x}_p \mathbin{\&} q_j) \multimap q_i \in \Sigma$$

where the top-right dots represent the induction hypothesis. For the top-left dots, we build a proof that can only succeed when Γ_v does not contains any occurrence of x_p, implementing a zero-test by proof-search. For $q \neq p < n$, we can remove the occurrence of x_q in Δ, x_q using $\overline{x}_p$ as a goal:

$$\frac{\dfrac{}{!\,\Sigma, x_q \vdash x_q}\ \mathrm{Ax} \qquad \dfrac{\cdots}{!\,\Sigma, \Delta \vdash \overline{x}_p}}{!\,\Sigma, \Delta, x_q \vdash \overline{x}_p}\ x_q \multimap (\overline{x}_p \multimap \overline{x}_p) \in \Sigma$$

As Γ_v is only composed of variables x_q with $q \neq p$, we repeat the previous step until Γ_v becomes empty and we end up with $!\,\Sigma, \emptyset \vdash \overline{x}_p$ which is then proved by

$$\frac{\dfrac{}{!\,\Sigma, \overline{x}_p \vdash \overline{x}_p}\ \mathrm{Ax}}{!\,\Sigma, \emptyset \vdash \overline{x}_p}\ (\overline{x}_p \multimap \overline{x}_p) \multimap \overline{x}_p \in \Sigma$$

This conclude the *only if part* of the argument.

The *if part* is obtained using soundness of trivial phase semantics (see Theorem 8.1). To simplify notations, we denote by $(\vec{x}_0, \ldots, \vec{x}_{n-1})$ the canonical basis of $(\mathbb{N}^n, +, \vec{0})$, i.e. we write $\mathbb{N}^n = \mathbb{N}.\vec{x}_0 \oplus \cdots \oplus \mathbb{N}.\vec{x}_{n-1}$ and the correspondence $x_i \rightleftharpoons \vec{x}_i$ induces a monoid isomorphism between the multisets generated by $\{x_0, \ldots, x_{n-1}\}$ and $(\mathbb{N}^n, +, \vec{0})$. We define the following interpretation of logical variables in $(\mathbb{N}^n, +, \vec{0})$:

$$[\![x_i]\!] := \{\vec{x}_i\} \qquad [\![\overline{x}_i]\!] := \{\vec{x}_i\}^\perp = \{w \mid w[i] = 0\}$$
$$[\![q_i]\!] := \{w \in \mathbb{N}^n \mid (1, P) \mathbin{/\!/} (i, w) \rightsquigarrow (0, \vec{0})\}$$

and we show that $\vec{0} \in [\![\sigma]\!]$ for any $\sigma \in \Sigma$. Hence the identity $[\![!\,\Sigma]\!] = \{\vec{0}\}$ holds. Writing $v = \alpha_1.\vec{x}_1 + \cdots + \alpha_n.\vec{x}_n$ we deduce $[\![\Gamma_v]\!] = [\![\alpha_1.x_1, \ldots, \alpha_n.x_n]\!] = \alpha_1.\{\vec{x}_1\} + \cdots + \alpha_n.\{\vec{x}_n\} = \{v\}$. As a consequence, we get $[\![!\,\Sigma, \Gamma_v]\!] = [\![!\,\Sigma]\!] + [\![\Gamma_v]\!] = \{v\}$. Thus, if $!\,\Sigma, \Gamma_v \vdash q_i$ has a proof in $\mathrm{G}_{\mathsf{elLL}}$, by soundness of trivial phase semantics we deduce $v \in [\![!\,\Sigma, \Gamma_v]\!] \subseteq [\![q_i]\!]$, i.e. the computation $(1, P) \mathbin{/\!/} (i, v) \rightsquigarrow (0, \vec{0})$ terminates. □

This allows us to conclude:

Theorem 8.4. MM $\preceq$ elLL.

9 Conclusion

Our formalisation consists of about 5200 lines of code. Around 200 loc are needed for the definition and reductions concerning PCP. The generic compiler needs 1500 loc. We have 1600 loc for the BSM section, 900 loc for the MM section, and 1000 loc for the ILL section.

Related work. There is not much related work concerning formalised undecidability proofs. Forster, Heiter, and Smolka [10] reduce the halting problem for Turing machines via string rewriting systems to PCP and further on to problems concerning context-free grammars.

Forster, Kirst and Smolka [11] reduce PCP to validity, satisfiability and provability in both classical and intuitionistic first-order logic (FOL). They have undecidability proofs not requiring intermediate steps, in striking contrast to the involved formalisations needed to complete our reduction.[6] They also include the notion of enumerability, connect (non)-enumerability with (un)decidability and give enumerability proofs for PCP and provable formulas.

With regard to the formalisation of models of computations, there is slightly more related work, starting with a proof of the s_{mn} theorem by Zammit [40]. Forster and Smolka [14] formalise the call-by-value λ-calculus as model of computation in Coq. They constructively develop the standard theory of computation, but only consider undecidability problems related to the λ-calculus.

Larchey-Wendling [23] formalises another standard model of computation, that of μ-recursive functions, which are shown to be representable by guarded Coq-terms. In particular, total μ-recursive predicates $\mathbb{N}^k \to \mathbb{N} \to \mathbb{P}$ can be constructively reified into functions of type $\mathbb{N}^k \to \mathbb{N}$.

Asperti and Ricciotti [1] formalise multi-tape Turing machines in the proof assistant Matita and construct a universal Turing machine. Ciaffaglione [6] formalises Turing machines in Coq using coinductive definitions and proves the halting problem to be undecidable. Vinogradova [38] gives an abstract discussion of undecidability using Turing categories.

Norrish [32] formalises computability theory based on the full λ-calculus in HOL. Xu, Zhang, and Urban [39] formalise Turing machines in Isabelle/HOL, prove the undecidability of the halting problem and give a universal Turing machine based on a translation from partial recursive functions. Ramos et al. [33] formalise the undecidability of the halting problem of a simple functional language in PVS.

[6] As it is highly expressive, FOL can directly express a characteristic formula for PCP. On the opposite side, directly giving a characterising ILL formula is quite infeasible, making intermediate steps crucial.

Forster and Kunze [12] present a framework to extract Coq functions to the call-by-value λ-calculus from [14] in a verified way. We could use this framework to prove the computability of all reductions explicitly, allowing for the assumption of non-constructive axioms, but thereby committing to a fixed model of computation.

Future work. We envision a library of undecidable problems in Coq. The library should contain basic problems that can be used for further reductions, like PCP, BSM, MM, and Halt as well as advanced problems with non-trivial proofs like ILL. We want to add the μ-recursive functions from [23], the halting problem for the call-by-value λ-calculus from [14], and more computational models, like for instance the WHILE language used by Jones [18]. Other basic problems that can serve as stepping stones for other reductions include tiling problems from e.g. Berger [2].

Formalising Hilbert's 10th problem — the undecidability of diophantine equations, — could be a challenge, e.g. following [17]. Finally, concerning linear logic, there are still fragments where the decidability status is unclear or debated: the entailment of multiplicative and exponential linear logic (MELL) was claimed to be decidable in [3] but this proof and hence the result is highly contested, see e.g. [9, footnote 1] and [36, footnote 4] and [24]. We hope that using the library, future research in this direction can be formalised.

Acknowledgments

The work of the second author was partially supported by the TICAMORE project (ANR grant 16-CE91-0002).

References

[1] Andrea Asperti and Wilmer Ricciotti. 2015. A formalization of multi-tape Turing machines. *Theoret. Comput. Sci.* 603 (2015), 23–42.

[2] Robert Berger. 1966. *The undecidability of the domino problem.* Number 66 in Memoirs of the American Mathematical Society. American Mathematical Soc.

[3] Katalin Bimbó. 2015. The decidability of the intensional fragment of classical linear logic. *Theoret. Comput. Sci.* 597 (2015), 1–17.

[4] Thomas Braibant and Damien Pous. 2010. An efficient Coq tactic for deciding Kleene algebras. In *International Conference on Interactive Theorem Proving*. Springer, 163–178.

[5] Adam Chlipala. 2010. A verified compiler for an impure functional language. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 93–106.

[6] Alberto Ciaffaglione. 2016. Towards Turing computability via coinduction. *Sci. Comput. Programming* 126 (2016), 31–51.

[7] Christian Doczkal and Joachim Bard. 2018. Completeness and decidability of converse PDL in the constructive type theory of Coq. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM, 42–52.

[8] Christian Doczkal and Gert Smolka. 2016. Completeness and decidability results for CTL in constructive type theory. *J. Automat. Reason.* 56, 3 (2016), 343–365.

[9] Diego Figueira, Ranko Lazic, Jérôme Leroux, Filip Mazowiecki, and Grégoire Sutre. 2017. Polynomial-Space Completeness of Reachability for Succinct Branching VASS in Dimension One. In *ICALP 2017 (LIPIcs)*, Vol. 80. Schloss Dagstuhl, 119:1–14.

[10] Yannick Forster, Edith Heiter, and Gert Smolka. 2018. Verification of PCP-Related Computational Reductions in Coq. In *International Conference on Interactive Theorem Proving*. Springer, 253–269.

[11] Yannick Forster, Dominik Kirst, and Gert Smolka. 2019. On Synthetic Undecidability in Coq, with an Application to the Entscheidungsproblem. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM.

[12] Yannick Forster and Fabian Kunze. 2016. Verified Extraction from Coq to a Lambda-Calculus. In *Coq Workshop 2016*.

[13] Yannick Forster and Dominique Larchey-Wendling. 2018. Towards a library of formalised undecidable problems in Coq: The undecidability of intuitionistic linear logic. *Workshop on Syntax and Semantics of Low-level Languages, Oxford* (2018).

[14] Yannick Forster and Gert Smolka. 2017. Weak call-by-value lambda calculus as a model of computation in Coq. In *Interactive Theorem Proving - 8th International Conference, ITP*. Springer, 189–206.

[15] Jean-Yves Girard. 1987. Linear logic. *Theoret. Comput. Sci.* 50, 1 (1987), 1–102.

[16] Jean-Yves Girard, Yves Lafont, and Laurent Regnier (Eds.). 1995. *Advances in Linear Logic*. London Mathematical Society Lecture Note Series, Vol. 222. Cambridge University Press.

[17] James P. Jones and Yuri V. Matijasevič. 1984. Register Machine Proof of the Theorem on Exponential Diophantine Representation of Enumerable Sets. *J. Symb. Log.* 49, 3 (1984), 818–829.

[18] Neil D. Jones. 1997. *Computability and complexity: from a programming perspective*. Vol. 21. MIT press.

[19] Max Kanovich. 1994. Linear Logic as a Logic of Computations. *Ann. Pure Appl. Logic* 67, 1–3 (1994), 183–212.

[20] Max Kanovich. 1995. The direct simulation of Minsky machines in linear logic. See [16], Chapter 2, 123–145.

[21] Chantal Keller and Thorsten Altenkirch. 2010. Hereditary substitutions for simple types, formalized. In *Proceedings of the third ACM SIGPLAN workshop on Mathematically structured functional programming*. ACM, 3–10.

[22] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 179–192.

[23] Dominique Larchey-Wendling. 2017. Typing Total Recursive Functions in Coq. In *Interactive Theorem Proving - 8th International Conference, ITP*. Springer, 371–388.

[24] Dominique Larchey-Wendling. 2018. Constructive Decision via Redundancy-Free Proof-Search. In *Automated Reasoning - 9th International Joint Conference, IJCAR, Proceedings*. Springer, 422–438.

[25] Dominique Larchey-Wendling and Didier Galmiche. 2010. The Undecidability of Boolean BI through Phase Semantics. In *LICS 2010*. IEEE Computer Society, 140–149.

[26] Dominique Larchey-Wendling and Didier Galmiche. 2013. Non-deterministic Phase Semantics and the Undecidability of Boolean BI. *ACM Trans. Comput. Log.* 14, 1 (2013), 6:1–6:41.

[27] Xavier Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 42–54.

[28] Patrick Lincoln. 1995. Deciding provability of linear logic formulas. See [16], Chapter 2, 109–122.

[29] Patrick Lincoln, John C. Mitchell, Andre Scedrov, and Natarajan Shankar. 1990. Decision Problems for Propositional Linear Logic. In *31st Annual Symposium on Foundations of Computer Science*, Vol. 2. IEEE Computer Society, 662–671.

[30] Petar Maksimović and Alan Schmitt. 2015. HOCore in Coq. In *International Conference on Interactive Theorem Proving*. Springer, 278–293.

[31] Marvin L. Minsky. 1967. *Computation: finite and infinite machines*. Prentice-Hall, Inc.

[32] Michael Norrish. 2011. Mechanised computability theory. In *International Conference on Interactive Theorem Proving*. Springer, 297–311.

[33] Thiago Mendonça Ferreira Ramos, César Muñoz, Mauricio Ayala-Rincón, Mariano Moscato, Aaron Dutle, and Anthony Narkawicz. 2018. Formalization of the Undecidability of the Halting Problem for a Functional Language. In *International Workshop on Logic, Language, Information, and Computation*. Springer, 196–209.

[34] Hartley Rogers. 1967. *Theory of recursive functions and effective computability*. Vol. 5. McGraw-Hill New York.

[35] Steven Schäfer, Gert Smolka, and Tobias Tebbi. 2015. Completeness and decidability of de Bruijn substitution algebra in Coq. In *Proceedings of the 2015 Conference on Certified Programs and Proofs*. ACM, 67–73.

[36] Sylvain Schmitz. 2016. The Complexity of Reachability in Vector Addition Systems. *ACM SIGLOG News* 3, 1 (2016), 4–21.

[37] The Coq proof assistant. 2018. http://coq.inria.fr. (2018).

[38] Polina Vinogradova. 2017. *Formalizing Abstract Computability: Turing Categories in Coq*. Ph.D. Dissertation. University of Ottawa.

[39] Jian Xu, Xingyuan Zhang, and Christian Urban. 2013. Mechanising turing machines and computability theory in Isabelle/HOL. In *International Conference on Interactive Theorem Proving*. Springer, 147–162.

[40] Vincent Zammit. 1997. *A Proof of the S-m-n theorem in Coq*. Technical report. The Computing Laboratory, The University of Kent, Canterbury, Kent, UK. http://kar.kent.ac.uk/21524/