Limitations of Program Verification

Mogens Nielsen

Computer Science Department
University of Aarhus, Denmark
dBerLog Lecture Notes, September 2007

1 Introduction

The purpose of this note is to prove formally some strong limitations of *any* formalization of the process of proving the correctness of computer programs.

The note builds on top of [dADS], and assumes familiarity with with propositional and predicate logic as introduced in [Kel], as well as basic computability theory [Mar]. Formally, we shall prove that there cannot exist any sound and complete proof system for the correctness of algorithms with input and output specifications as defined in [dADS]. This result will follow from a proof that the set of correct program specifications is not a recursively enumerable set, whereas the set of provable program specifications will be shown to be enumerable for any reasonable notion of provability.

As a corollary to our proof, we shall present a simple proof of one of the major results on the foundations of mathematics from the 20'th century: Gödel's incompleteness theorem, and discuss some of its implications for computer science.

The structure of the note is as follows. In the next three sections we formalise the process of program verification from [dADS], and after a small technical section on strings and numbers, we conclude with two sections presenting our two main incompleteness results.

2 Programs and Specifications

We recall the notion of program specifications (as preconditions and postconditions) and program correctness from [dADS], and we formalize these notions in terms of Hoare triples for a simple programming language PLN.

The purpose of [dADS] is to show how one can formally reason about the correctness of programs/methods by means of syntactic "proof obligations". [dADS] contains a number of elegant proofs, based on clever logical formulations of e.g.

invariants of while-loops. Even for small programs the task of proving correctness is cumbersome, since most of the work is involved is routine logical reasoning, which seems to be better suited for automated rather than human reasoning. However, as we shall prove in the following, there is *no* way of automising the process of proving the correctness of programs. In the popular literature, this is often interpreted as saying, that there is no automated way of replacing the human creativity in guessing invariants of while-loops etc. in the process of program verification, as illustrated in [dADS] – any automated tool will either prove specifications which are not correct, or will fail to prove some correct specifications.

We shall show this negative result even for a very simple instance of the algorithmic language from [dADS]. The syntax of the language will basically be that of [dADS], but we shall restrict ourselves to a language, where variables can only take natural numbers $N = \{0, 1, 2, \ldots\}$ as values. So, to be more specific, the syntax of our complete language, which we call PLN - Programming Language for Natural numbers, will be as shown in Figure 1. Notice that in order to avoid confusion between the symbol $\leftarrow$ used in [dADS] for assignment and in [Kel] for logical implication, we choose to use $:=$ for assignments in PLN.

$$
\begin{array}{lll}
Con & ::= & \mathsf{0} \mid \mathsf{1} \mid \mathsf{2} \mid \ldots \\
Var & ::= & \mathsf{x} \mid \mathsf{y} \mid \mathsf{z} \mid \ldots \\
E & ::= & Con \mid Var \mid E + E \mid E * E \mid (E) \\
B & ::= & \mathsf{true} \mid \mathsf{false} \mid \neg B \mid B \wedge B \mid B \vee B \mid E = E \mid (B) \\
C & ::= & Var := E \mid C; C \mid \mathsf{if}\ B\ \mathsf{then}\ C\ \mathsf{else}\ \ C \mid \mathsf{while}\ B\ \mathsf{do}\ C
\end{array}
$$

Fig. 1. Syntax of PLN

A PLN program is simply a PLN command, C. With respect to the operational semantics of PLN we refer to [dADS], where it is defined formally in terms of a transition system over configurations $[C, \sigma]$, where C is a command, and σ is a state assigning values to program variables, i.e. $\sigma : Var \to N$, and with a transition relation $\triangleright$ repesenting "small step execution". The operational semantics associates with any PLN program, C, a partial function $Sem[C] : States \rightharpoonup States$, such that for any program state, σ, $Sem[C](\sigma)$ is undefined if C diverges (does not halt) when started in σ, and $Sem[C](\sigma) = \sigma'$, if C when started in state σ terminates in state σ', i.e. $[C, \sigma] \triangleright^* \sigma'$.

Example 1. Consider the following PLN program, $C_{\ominus}$:

$$
\begin{array}{l}
\mathsf{y} := \mathsf{0}; \\
\mathsf{while}\ \neg(\mathsf{y} = \mathsf{m} \vee \mathsf{y} = \mathsf{n})\ \mathsf{do}\ \mathsf{y} := \mathsf{y} + \mathsf{1}; \\
\mathsf{x} := \mathsf{0}; \\
\mathsf{while}\ \neg(\mathsf{y} = \mathsf{m})\ \mathsf{do}\ \mathsf{y} := \mathsf{y} + \mathsf{1}; \mathsf{x} := \mathsf{x} + \mathsf{1}
\end{array}
$$

If $C_\ominus$ is started in a state σ, for which $\sigma(\mathsf{m}) = 7$ and $\sigma(\mathsf{n}) = 5$, then $C_\ominus$ terminates in a state σ' with with $\sigma'(\mathsf{x}) = 7 - 5 = 2$. If $C_\ominus$ is started in a state σ with $\sigma(\mathsf{m}) = 3$ and $\sigma(\mathsf{n}) = 4$, then $C_\ominus$ terminates in a state σ' with $\sigma'(\mathsf{x}) = 0$. You may convince yourself that $C_\ominus$ computes in x the natural number version of "m minus n" (usually called "m monus n"):

$$m \ominus n = \begin{cases} m - n & \text{if } m > n \\ 0 & \text{if } m \leq n \end{cases}$$

As an example, $Sem[C_\ominus](\mathsf{m} = 7, \mathsf{n} = 5, \ldots) = (\mathsf{m} = 7, \mathsf{n} = 5, \mathsf{x} = 2, \mathsf{y} = 7, \ldots)$ □

The correctness of program/method $C_\ominus$ would be expressed in [dADS] by "decorating" $C_\ominus$ with input- and output-state specifications, expressed in some suitable logical language.

Our choice of such a language will be exactly the first order language of predicate logic as defined in [Kel]. Following [Kel] we shall use the symbol $\bot$ standing for the constant predicate "always false", i.e. the predicate which is false in any interpretation, and similarly we shall use the symbol $\top$ standing for the constant predicate "always true", i.e. the predicate which is true in any interpretation.

A natural specification for $C_\ominus$ would be to specify that for any input state, $C_\ominus$ terminates in a state for which the value of x is equal to $\mathsf{m} \ominus \mathsf{n}$, i.e.

$\{\top\}\ C_\ominus\ \{\mathsf{x} = \mathsf{m} \ominus \mathsf{n}\}$.

In the literature such a correctness specification is usually called a Hoare triple. Following [Kel] we can now be more formal and specific with respect to the logical language of state specifications. It is clearly a logical language interpreted over (program) variables, so the most natural thing is to formalise the language as first order predicate logic interpreted in the model N of natural numbers.

Notice that the satisfaction of state specifications like $\{\mathsf{x} = \mathsf{m} \ominus \mathsf{n}\}$ is precisely captured by the formal definition of the formula interpreted in the model of natural numbers as defined in [Kel], where the terminology "valuation" is used in stead of "state" used in [dADS]. Formally, an "valuation" is a slight generalization of "state", in the sense that an "valuation" may refer to other variables than just program variables (notice that this happens frequently in [dADS]).

So, using the terminology from [Kel] we write

$N \models_\sigma \varphi$

as notation for the fact that the state σ satisfies φ in the interpretation of natural numbers N. As examples, for $\sigma(\mathsf{x}) = 3$, $\sigma(\mathsf{m}) = 8$, and $\sigma(\mathsf{n}) = 5$ we have

$N \models_\sigma \mathsf{x} = \mathsf{m} \ominus \mathsf{n}$, but
$N \not\models_\sigma \mathsf{x} > \mathsf{m} \vee \mathsf{x} > \mathsf{n}$.

With this definition, we may now formalise the notions of partial and total correctness from [dADS]:

Definition 1. *Let C be a PLN program, and φ and ψ be N formulae. Then the Hoare triple $\{\varphi\}C\{\psi\}$ is said to be true under partial correctness,*

$\models_{par} \{\varphi\}C\{\psi\}$,

iff

for all states σ, if $N \models_\sigma \varphi$ and $Sem[C](\sigma)$ is defined and equal to σ', then $N \models_{\sigma'} \psi$,

i.e. for all states σ, satisfying φ, if C terminates when started in state σ, then the final state satisfies ψ.

$\{\varphi\}C\{\psi\}$ is said to be true under total correctness,

$\models_{tot} \{\varphi\}C\{\psi\}$,

iff

for all states σ, if $N \models_\sigma \varphi$, then $Sem[C](\sigma)$ is defined, and if $Sem[C](\sigma) = \sigma'$ then $N \models_{\sigma'} \psi$,

i.e. for all states σ, satisfying φ, C terminates when started in state σ, and the final state satisfies ψ. □

So, we may now formally express the intended correctness of $C_\ominus$ as ($C_\ominus$ always terminates, and always ends up with the value of x equal to $\mathsf{m} \ominus \mathsf{n}$):

$\models_{tot} \{\top\}C_\ominus\{\mathsf{x} = \mathsf{m} \ominus \mathsf{n}\}$.

2.1 *PLN* expressiveness

We have formalised the fundamental notions of partial and total correctness for a very simple programming language, PLN. We end this section with a few observations and examples showing that PLN despite its simplicity is capable of expressing some primitives, which we shall find useful later on (exponentiation and integer division).

Given that you can compute the monus value of any two program variables, it should be clear (convince yourself how) that we may also express in PLN the operators $\leq$ and $>$ in PLN, since

$m \leq n$ iff $m \ominus n = 0$, and
$m > n$ iff $\neg(m \leq n)$.

Example 2. Consider the following program $C_{\uparrow}$:

$\mathsf{x} := 1; \mathsf{y} := 0;$
$\mathsf{while}\ \neg(\mathsf{y} = \mathsf{n})\ \mathsf{do}\ (\mathsf{x} := \mathsf{x} * \mathsf{m};\ \mathsf{y} := \mathsf{y} + 1)$

The fact that this PLN program is a correct implementation of the operation of exponentiation, can be expressed (and proved using the techniques from [dADS]), as follows:

$$\models_{tot} \{\top\}\ C_{\uparrow}\ \{\mathsf{x} = \mathsf{m}^{\mathsf{n}}\}. \qquad \square$$

Example 3. Let $div(m, n)$ denote "integer division of m by n" and $rem(m, n)$ denote "remainder of integer division of m by n" (e.g. $div(14, 4) = 3$ and $rem(14, 4) = 2$). Formally $div(m, n)$ and $rem(m, n)$ are defined as the unique numbers satisfying $m = n * div(m, n) + rem(m, n)$, where $0 \leq rem(m, n) < n$. The fact that integer division and remainder can be expressed in PLN can now be expressed as:

$$\models_{tot} \{\neg(\mathsf{n} = 0)\}\ C_{div}\ \{\mathsf{d} = div(\mathsf{m}, \mathsf{n}) \wedge \mathsf{r} = rem(\mathsf{m},\ \mathsf{n})\}. \qquad \square$$

where C_{div} is the following PLN program:

$\mathsf{d} := 0; \mathsf{r} := \mathsf{m};$
$\mathsf{while}\ \mathsf{r} \geq \mathsf{n}\ \mathsf{do}\ (\mathsf{r} := \mathsf{r} \ominus \mathsf{n};\ \mathsf{d} := \mathsf{d} + 1)$

3 Verification

We recall the program verification methodology (based on proof obligations) from [dADS], and we formalize this methodology in terms of a proof system for Hoare triples and the simple programming language PLN.

In the previous section we formalised the notions of partial and total correctness from [dADS] in the form of a logical language of Hoare triples: $\models_{par} \{\varphi\}C\{\psi\}$ (or $\models_{tot} \{\varphi\}C\{\psi\}$). In this section we shall go one step further and attempt to formalise also the the line of reasoning introduced in [dADS] for proving the correctness of program specifications. More specifically, we shall illustrate how the *algorithmic proof principles* from [dADS] can be formalized as a formal proof system in the spirit of an axiomatic system as defined in [Kel].

In [Kel] the axiomatic proof system AL for propositional logic is proved to be sound and complete. The extended system FOPL for first order predicate logic in [Kel] is sound and complete (although this is not proved in [Kel]). So a good question is whether one can construct a sound and complete proof system for Hoare triples $\models_{par} \{\varphi\}C\{\psi\}$ (or $\models_{tot} \{\varphi\}C\{\psi\}$), that is a proof system proving exactly the correctly annotated PLN programs.

In Figure 2 below we present a set of proof rules (the so-called Hoare rules) with the property that they express precisely the reasoning techniques employed in [dADS] in terms of a formal proof system. The system is nothing but the axiomatic system for programming introduced in [Kel] chapter 7 with the following modifications.

For simplicity, we present the system here in terms of one axiom and four rules of deduction. Secondly, we use a more standard notation for axioms and rules of deduction, which should be read as follows: if you can prove what is above the line of a rule (the premisses), then you may derive what is below the line (the conclusion). So, an axiom is a rule without premisses. Finally, following [Kel] we use the notation $\vdash_{par} \{\varphi\}C\{\psi\}$ to denote, that of the triple $\{\varphi\}C\{\psi\}$ can be deduced/derived using the Hoare rules.

$$\frac{}{\{\varphi[E/\mathsf{x}]\}\ \mathsf{x} := E\ \{\varphi\}}\ \text{Ass-axiom} \qquad \frac{\{\varphi\}C_1\{\eta\} \quad \{\eta\}C_2\{\psi\}}{\{\varphi\}C_1; C_2\{\psi\}}\ \text{Comp-rule}$$

$$\frac{\{\varphi \wedge B\}C_1\{\psi\} \quad \{\varphi \wedge \neg B\}C_2\{\psi\}}{\{\varphi\}\ \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \{\psi\}}\ \text{If-rule} \qquad \frac{\{\varphi \wedge B\}C\{\varphi\}}{\{\varphi\}\ \mathsf{while}\ B\ \mathsf{do}\ C\ \{\varphi\ \wedge \neg B\}}\ \text{While-rule}$$

$$\frac{\vdash_N \varphi' \to \varphi \quad \{\varphi\}C\{\psi\} \quad \vdash_N \psi \to \psi'}{\{\varphi'\}C\{\psi'\}}\ \text{Impl-rule}$$

Fig. 2. Hoare rules for partial correctness $\vdash_{par} \{\varphi\}C\{\psi\}$

Let us explain the rules by constructing a proof of the partial correctness of Euclid's algorithm from [dADS], page 33, computing the greatest common divisor (gcd) of two natural numbers:

$$\vdash_{par} \{\mathsf{m} = m_0 \geq 1 \wedge \mathsf{n} = n_0 \geq 1\}\ P_{Euc}\ \{\mathsf{r} = gcd(m_0, n_0)\}$$

where P_{Euc} is the following PLN program:

$$\mathsf{while}\ \neg(\mathsf{m} = \mathsf{n})\ \mathsf{do}\ \mathsf{if}\ \mathsf{m} > \mathsf{n}\ \mathsf{then}\ \mathsf{m} := \mathsf{m} - \mathsf{n}\ \mathsf{else}\ \mathsf{n} := \mathsf{n} - \mathsf{m};$$
$$\mathsf{r} := \mathsf{m}$$

Notice that m, n and r in the state specifications refer to program variables, whereas m_0 and n_0 are logical variables referring to the initial values of program variables m, and n.

The proof is best explained "backwards", i.e. starting form the goal

$$\vdash_{par} \{\mathsf{m} = m_0 \geq 1 \wedge \mathsf{n} = n_0 \geq 1\}P_{Euc}\{\mathsf{r} = gcd(m_0, n_0)\}.$$

P_{Euc} is a sequential composition of a while-loop and an assignment, and hence in order to prove our P_{Euc} specification we may use the Comp-rule which has exactly the desired conclusion with (here and in the following we use the

symbol $\equiv$ for equality between formulae, in order not to create any confusion with the symbol $=$ used within formulae)

$\varphi \equiv \mathsf{m} = m_0 \geq 1 \wedge \mathsf{n} = n_0 \geq 1$, and
$\psi \equiv \mathsf{r} = gcd(m_0, n_0)$.

This involves inventing a suitable intermediate formula η, intended to express a property satisfied by the state immediately before executing the assignment $\mathsf{r} := \mathsf{m}$. Applying the Comp-rule, we reduce our proof obligation to the two (simpler) proof obligations in the premises of the application of the Comp-rule:

$\{\mathsf{m} = m_0 \geq 1 \wedge \mathsf{n} = n_0 \geq 1\}$
$\mathsf{while}\ \neg(\mathsf{m} = \mathsf{n})\ \mathsf{do\ if}\ \mathsf{m} > \mathsf{n}\ \mathsf{then}\ \mathsf{m} := \mathsf{m} - \mathsf{n}\ \mathsf{else}\ \mathsf{n} := \mathsf{n} - \mathsf{m}$
$\{\eta\}$

and

$\{\eta\}\ \mathsf{r} := \mathsf{m}\ \{\mathsf{r} = gcd(m_0, n_0)\}$

In the second proof obligation, the program is a simple assignment, and hence we may attempt to use the Ass-axiom:

$$\frac{}{\{\varphi[E/\mathsf{x}]\}\ \mathsf{x} := E\ \{\varphi\}}$$

This axiom may look a little strange, but thinking about it in the following way, should explain the intuition behind the rule. Assume that φ holds in the state following the assignment $\mathsf{x} := E$, what can you say about the state immediately proceeding the assignment? Well, the only change which has happened is that the variable x has been assigned a new value - all other variables are left unchanged. That is evaluating the boolean value of φ after the assigment, will refer to the value of x as precisely the value of the expression E in the state before the assignment. And hence a necessary and sufficient condition for φ being true after the assignment is that evaluating φ in the proceeding state with the value of E substituted for x should evaluate to true. Applying this rule to $\mathsf{r} := \mathsf{m}$ and the post-condition $\mathsf{r} = gcd(m_0, n_0)$ we get the precondition

$$(\mathsf{r} = gcd(m_0, n_0))[\mathsf{m}/\mathsf{r}] \equiv (\mathsf{m} = gcd(m_0, n_0)),$$

i.e. a necessary and sufficient condition for $\mathsf{r} = gcd(m_0, n_0)$ to hold after the assignment $\mathsf{r} := \mathsf{m}$ is that $\mathsf{m} = gcd(m_0, n_0)$ holds before the assignment, which (hopefully) makes perfect sense. Notice that the Ass-axiom is an axiom in our proof system (no premisses), and applying it the way we just did "from right to left" it is completely mechanical - we simply substitute all occurrences of x in φ by E!

So, by using the Ass-axiom, we have a proof in our proof system of

$\vdash_{par} \{\mathsf{m} = gcd(m_0, n_0)\}\ \mathsf{r} := \mathsf{m}\ \{\mathsf{r} = gcd(m_0, n_0)\}$,

and hence if we choose

$$\eta \equiv \ \mathsf{m} = gcd(m_0, n_0)$$

we are left with only one proof obligation:

$$\{\mathsf{m} = m_0 \geq 1 \wedge \mathsf{n} = n_0 \geq 1\}$$
$$\mathsf{while}\ \neg(\mathsf{m} = \mathsf{n})\ \mathsf{do\ if}\ \mathsf{m} > \mathsf{n}\ \mathsf{then}\ \mathsf{m} := \mathsf{m} - \mathsf{n}\ \mathsf{else}\ \mathsf{n} := \mathsf{n} - \mathsf{m}$$
$$\{\mathsf{m} = gcd(m_0, n_0)\}.$$

The program part of this triple is a while-loop, and hence we may attempt to apply the While-rule:

$$\frac{\{\varphi \wedge B\}C\{\varphi\}}{\{\varphi\}\ \mathsf{while}\ B\ \mathsf{do}\ C\ \{\varphi\ \wedge \neg B\}}$$

In order to apply this rule, we need to invent an invariant φ. Following [dADS] we are lead to choose the invariant $\varphi \equiv \ gcd(\mathsf{m}, \mathsf{n}) = gcd(m_0, n_0)$, which from a proof of

$$\{gcd(\mathsf{m}, \mathsf{n}) = gcd(m_0, n_0) \wedge \neg(\mathsf{m} = \mathsf{n})\}$$
$$\mathsf{if}\ \mathsf{m} > \mathsf{n}\ \mathsf{then}\ \mathsf{m} := \mathsf{m} - \mathsf{n}\ \mathsf{else}\ \mathsf{n} := \mathsf{n} - \mathsf{m}$$
$$\{gcd(\mathsf{m}, \mathsf{n}) = gcd(m_0, n_0)\}$$

will allow us to conclude using the While-rule

$$\vdash_{par} \{gcd(\mathsf{m}, \mathsf{n}) = gcd(m_0, n_0)\}$$
$$\mathsf{while}\ \neg(\mathsf{m} = \mathsf{n})\ \mathsf{do\ if}\ \mathsf{m} > \mathsf{n}\ \mathsf{then}\ \mathsf{m} := \mathsf{m} - \mathsf{n}\ \mathsf{else}\ \mathsf{n} := \mathsf{n} - \mathsf{m}$$
$$\{gcd(\mathsf{m}, \mathsf{n}) = gcd(m_0, n_0) \wedge \neg\neg(\mathsf{m} = \mathsf{n})\}.$$

Unfortunately, the conclusion of this rule does not quite match our proof obligation. And this is where the Impl-rule comes into play. Before applying the rule, let us comment on how to read the rule and argue for the soundness of it:

$$\frac{\vdash_N \varphi' \rightarrow \varphi \quad \{\varphi\}C\{\psi\} \quad \vdash_N \psi \rightarrow \psi'}{\{\varphi'\}C\{\psi'\}}$$

The rule should be read as follows. Assume that you have a proof (in some unspecified specified proof system $\vdash_N$) of the properties $\varphi' \rightarrow \varphi$ and $\psi \rightarrow \psi'$ in N , and that you have a proof using the Hoare rules of $\vdash_{par} \{\varphi\}C\{\psi\}$, then the rule allows you construct a proof of $\vdash_{par} \{\varphi'\}C\{\psi'\}$. We deliberately leave the proof system for $\vdash_N$ unspecified for the time being – we shall return to the existence of such a system in great detail later. For the time being we just assume that the proof system for $\vdash_N$ is sound, in the sense that any formula η which can be proved is semantically true in the interpretation of natural numbers, i.e. if $\vdash_N \eta$ then $N \models \eta$.

The soundness of the Impl-rule will now follow from the following arguments. Assume that the premisses of the rule hold semantically, i.e.

$N \models \varphi' \to \varphi$, $\models_{par} \{\varphi\}C\{\psi\}$, and $N \models \psi \to \psi'$

$\models_{par} \{\varphi\}C\{\psi\}$ semantically means that whenever the program C is started in a state satisfying φ, if it terminates, the final state will satisfy ψ. But from $N \models \varphi' \to \varphi$, any state satisfying φ' also satisfies φ, and similarly any state satisfying ψ satisfies ψ', and hence whenever the program C is started in a state satisfying φ', if it terminates, the final state will satisfy ψ', – the conclusion of the rule.

Applying the Impl-rule to our concrete

$\vdash_{par} \{\varphi \equiv gcd(\mathsf{m},\mathsf{n}) = gcd(m_0, n_0)\}$
$\mathsf{while}\ \neg(\mathsf{m} = \mathsf{n})\ \mathsf{do\ if}\ \mathsf{m} > \mathsf{n}\ \mathsf{then}\ \mathsf{m} := \mathsf{m} - \mathsf{n}\ \mathsf{else}\ \mathsf{n} := \mathsf{n} - \mathsf{m}$
$\{\psi \equiv gcd(\mathsf{m},\mathsf{n}) = gcd(m_0, n_0) \wedge \neg\neg(\mathsf{m} = \mathsf{n})\}$

we see that we may conclude our desired

$\vdash_{par} \{\varphi' \equiv \mathsf{m} = m_0 \geq 1 \wedge \mathsf{n} = n_0 \geq 1\}$
$\mathsf{while}\ \neg(\mathsf{m} = \mathsf{n})\ \mathsf{do\ if}\ \mathsf{m} > \mathsf{n}\ \mathsf{then}\ \mathsf{m} := \mathsf{m} - \mathsf{n}\ \mathsf{else}\ \mathsf{n} := \mathsf{n} - \mathsf{m}$
$\{\psi' \equiv \mathsf{m} = gcd(m_0, n_0)\}$.

from proofs of the following facts in the interpretation of natural numbers

$\vdash_N (\mathsf{m} = m_0 \geq 1 \wedge \mathsf{n} = n_0 \geq 1) \to gcd(\mathsf{m},\mathsf{n}) = gcd(m_0, n_0)$, and
$\vdash_N (gcd(\mathsf{m},\mathsf{n}) = gcd(m_0, n_0) \wedge \neg\neg(\mathsf{m} = \mathsf{n})) \to \mathsf{m} = gcd(m_0, n_0)$.

As mentioned we have not included any rules for proving formulae of the form $\vdash_N \varphi \to \psi$. In [dADS] proof obligations of this type are carried out by some semantic arguments based on properties of natural numbers, and in [dADS] you may find detailed arguments for the required properties of *gcd*. For the purpose of this section, we simply assume that we have some proof system available formalising the kind of reasoning about natural numbers employed in [dADS] (we shall later see a proof of the fact that actually no such sound and complete proof system can exist!).

Given these assumptions, our proof of P_{Euc} is reduced to the proof obligation:

$\{gcd(m,n) = gcd(m_0, n_0) \wedge \neg(m = n)\}$
$\mathsf{if}\ \mathsf{m} > \mathsf{n}\ \mathsf{then}\ \mathsf{m} := \mathsf{m} - \mathsf{n}\ \mathsf{else}\ \mathsf{n} := \mathsf{n} - \mathsf{m}$
$\{gcd(m,n) = gcd(m_0, n_0)\}$

By now it should be clear how to proceed by applying the If-rule, reducing our proof obligation to the following two obligations:

$\{gcd(\mathsf{m},\mathsf{n}) = gcd(m_0, n_0) \wedge \neg(\mathsf{m} = \mathsf{n}) \wedge (\mathsf{m} > \mathsf{n})\}$
$\mathsf{m} := \mathsf{m} - \mathsf{n}$
$\{gcd(\mathsf{m},\mathsf{n}) = gcd(m_0, n_0)\}$

$\{gcd(\mathsf{m},\mathsf{n}) = gcd(m_0,n_0) \wedge \neg(\mathsf{m}=\mathsf{n}) \wedge \neg(\mathsf{m}>\mathsf{n})\}$
$\mathsf{n} := \mathsf{n} - \mathsf{m}$
$\{gcd(\mathsf{m},\mathsf{n}) = gcd(m_0,n_0)\}$

Applying the Ass-axiom to the first of these obligations gives us a proof of

$\vdash_{par} \{gcd(\mathsf{m}-\mathsf{n},\mathsf{n}) = gcd(m_0,n_0)\}$
$\mathsf{m} := \mathsf{m} - \mathsf{n}$
$\{gcd(\mathsf{m},\mathsf{n}) = gcd(m_0,n_0)\}$

and using the Impl rule gives us a proof of our desired proof obligation from the following fact of natural numbers, for which you may again find a proof in [dADS]

$\vdash_N (gcd(\mathsf{m},\mathsf{n}) = gcd(m_0,n_0) \wedge \neg(\mathsf{m}=\mathsf{n}) \wedge \neg(\mathsf{m}>\mathsf{n}))$
$\rightarrow\ gcd(\mathsf{m}-\mathsf{n},\mathsf{n}) = gcd(m_0,n_0).$

The second and final proof obligation above is proved symmetrically. A summary of our complete proof is provided in Figure 3 below, following the style of presenting proofs in [Kel].

Exercise 1. Prove formally the correctness specification of the exponentiation program $C_\uparrow$ from Example 2 using the Hoare rules. Whenever applying the Impl-rule with proof obligations of the form $\vdash_N \varphi \rightarrow \psi$ argue informally the the required formulae are true in the model of natural numbers (as in the proof of $P_\ominus$ above). □

4 Soundness and Completeness

We discuss the notion of a sound and complete proof system for Hoare triples over PLN, and we prepare for our main incompleteness results by showing the set of provable formulae (theorems) for any proof system is recursively enumerable.

Let us now make precise what is the formal definition of soundness and completeness of proof systems for Hoare triples.

Definition 2. *Assume we have some proof system for Hoare triples defining a set of provable assertions of the form $\vdash_{par} \{\varphi\}C\{\psi\}$, where ψ and φ are N formulae, and C is a PLN program.*

*Such a proof system is said to be **sound** if every provable assertion is also true (under partial correctness), i.e.*

1	$\vdash_N$	$gcd(\mathsf{m},\mathsf{n}) = gcd(m_0,n_0) \wedge \neg(\mathsf{m}=\mathsf{n}) \wedge (\mathsf{m}>\mathsf{n})$	Assumed
		$\rightarrow gcd(\mathsf{m}-\mathsf{n},\mathsf{n}) = gcd(m_0,n_0)$	$\vdash_N$ proof
2	$\vdash_{par}$	$\{gcd(\mathsf{m}-\mathsf{n},\mathsf{n}) = gcd(m_0,n_0)\}$	
		$\mathsf{m} := \mathsf{m}-\mathsf{n}$	Ass-axiom
		$\{gcd(\mathsf{m},\mathsf{n}) = gcd(m_0,n_0)\}$	
3	$\vdash_{par}$	$\{gcd(\mathsf{m},\mathsf{n}) = gcd(m_0,n_0) \wedge \neg(\mathsf{m}=\mathsf{n}) \wedge (\mathsf{m}>\mathsf{n})\}$	
		$\mathsf{m} := \mathsf{m}-\mathsf{n}$	Impl-rule 1,2
		$\{gcd(\mathsf{m},\mathsf{n}) = gcd(m_0,n_0)\}$	
4	$\vdash_N$	$gcd(\mathsf{m},\mathsf{n}) = gcd(m_0,n_0) \wedge \neg(\mathsf{m}=\mathsf{n}) \wedge \neg(\mathsf{m}>\mathsf{n})$	Assumed
		$\rightarrow gcd(\mathsf{m},\mathsf{n}-\mathsf{m}) = gcd(m_0,n_0)$	$\vdash_N$ proof
5	$\vdash_{par}$	$\{gcd(\mathsf{m},\mathsf{n}-\mathsf{m}) = gcd(m_0,n_0)\}$	
		$\mathsf{n} := \mathsf{n}-\mathsf{m}$	Ass-axiom
		$\{gcd(\mathsf{m},\mathsf{n}) = gcd(m_0,n_0)\}$	
6	$\vdash_{par}$	$\{gcd(\mathsf{m},\mathsf{n}) = gcd(m_0,n_0) \wedge \neg(\mathsf{m}=\mathsf{n}) \wedge \neg(\mathsf{m}>\mathsf{n}\}$	
		$\mathsf{n} := \mathsf{n}-\mathsf{m}$	Impl-rule 4,5
		$\{gcd(\mathsf{m},\mathsf{n}) = gcd(m_0,n_0)\}$	
7	$\vdash_{par}$	$\{gcd(\mathsf{m},\mathsf{n}) = gcd(m_0,n_0) \wedge \neg(\mathsf{m}=\mathsf{n})\}$	
		$\mathsf{if}\ \mathsf{m}>\mathsf{n}\ \mathsf{then}\ \mathsf{m} := \mathsf{m}-\mathsf{n}\ \mathsf{else}\ \mathsf{n} := \mathsf{n}-\mathsf{m}$	If-rule 3,6
		$\{gcd(\mathsf{m},\mathsf{n}) = gcd(m_0,n_0)\}$	
8	$\vdash_{par}$	$\{gcd(\mathsf{m},\mathsf{n}) = gcd(m_0,n_0)\}$	
		$\mathsf{while}\ \neg(\mathsf{m}=\mathsf{n})\ \mathsf{do}$	While-rule 7
		$\quad \mathsf{if}\ \mathsf{m}>\mathsf{n}\ \mathsf{then}\ \mathsf{m} := \mathsf{m}-\mathsf{n}\ \mathsf{else}\ \mathsf{n} := \mathsf{n}-\mathsf{m}$	
		$\{gcd(\mathsf{m},\mathsf{n}) = gcd(m_0,n_0) \wedge \neg\neg(\mathsf{m}=\mathsf{n})\}$	
9	$\vdash_N$	$\mathsf{m} = m_0 \geq 1 \wedge \mathsf{n} = n_0 \geq 1$	Assumed
		$\rightarrow gcd(\mathsf{m},\mathsf{n}) = gcd(m_0,n_0)$	$\vdash_N$ proof
10	$\vdash_N$	$gcd(\mathsf{m},\mathsf{n}) = gcd(m_0,n_0) \wedge \neg\neg(\mathsf{m}=\mathsf{n})$	Assumed
		$\rightarrow \mathsf{m} = gcd(m_0,n_0)$	$\vdash_N$ proof
11	$\vdash_{par}$	$\{\mathsf{m} = m_0 \geq 1 \wedge \mathsf{n} = n_0 \geq 1\}$	
		$\mathsf{while}\ \neg(\mathsf{m}=\mathsf{n})\ \mathsf{do}$	
		$\quad \mathsf{if}\ \mathsf{m}>\mathsf{n}\ \mathsf{then}\ \mathsf{m} := \mathsf{m}-\mathsf{n}\ \mathsf{else}\ \mathsf{n} := \mathsf{n}-\mathsf{m}$	Impl rule 8,9,10
		$\{\mathsf{m} = gcd(m_0,n_0)\}$	
12	$\vdash_{par}$	$\{\mathsf{m} = gcd(m_0,n_0)\}\ \mathsf{r} := \mathsf{m}\ \{\mathsf{r} = gcd(m_0,n_0)\}$	Ass-axiom
13	$\vdash_{par}$	$\{\mathsf{m} = m_0 \geq 1 \wedge \mathsf{n} = n_0 \geq 1\}$	
		$\mathsf{while}\ \neg(\mathsf{m}=\mathsf{n})\ \mathsf{do}$	
		$\quad \mathsf{if}\ \mathsf{m}>\mathsf{n}\ \mathsf{then}\ \mathsf{m} := \mathsf{m}-\mathsf{n}\ \mathsf{else}\ \mathsf{n} := \mathsf{n}-\mathsf{m};$	
		$\mathsf{r} := \mathsf{m}$	Comp-rule 11,12
		$\{\mathsf{r} = gcd(m_0,n_0)\}$	

Fig. 3. Proof of P_{Euc}

whenever $\vdash_{par} \{\varphi\}C\{\psi\}$ *then* $\models_{par} \{\varphi\}C\{\psi\}$.

The proof system is said to be ***complete*** *if every assertion which is true (under partial correctness) can also be proved, i.e*

whenever $\models_{par} \{\varphi\}C\{\psi\}$ *then* $\vdash_{par} \{\varphi\}C\{\psi\}$.

A similar definition applies to the notion of a sound and complete proof systems under total correctness. □

Is the set of Hoare rules a sound and complete proof system for Hoare triples? This question does not really make any sense, since we deliberately did not include any rules for proving properties of natural numbers used in the premises of the Impl-rule. In the proof of P_{Euc} all correctness assertions $\vdash_{par}$ are proved using the Hoare rules, whereas all $\vdash_N$ assertions are implicitly assumed to be proved in some other unspecified proof system for natural numbers. Now let us concentrate on the existence of such a proof system.

Our first choice could be the sound and complete proof rules for predicate logic from [Kel]. These rules are sound in the interpretation of natural numbers (since they are sound rules for formulae in *all* interpretations), but clearly they are not complete, i.e. there exists properties of natural numbers, which cannot be proved from these rules. Any property which relies on the fact that we are talking about the natural numbers and not some other interpretation of the constant symbols 0,1, and function symbols +, gcd etc. will not be provable in this system. One example of such a formula could be $\forall n(n < n+1)$. So, we need to extend our proof rules for predicate logic with rules proving all these extra formulae.

Before we even attempt to look for such rules, we need to decide on the exact vocabulary of constant symbols, function symbols, and predicate symbols we have in mind. Notice that we have been rather loose on this choice in our discussion of correctness proofs above, but in order to even ask the question of completeness, we need to be more specific.

As an example, let us consider the very simple set of constant symbols 0 and 1 interpreted as the natural numbers 'zero' and 'one' respectively, the function symbols $+, *$, and $\uparrow$ interpreted as addition, multiplication, and exponentiation respectively, and the predicate symbol $=$ interpreted as equality in N. One of the most popular set of rules attempting to achieve a sound and complete proof systems for predicate logic formulae over this vocabulary in the interpretation of natural numbers is that of so-called Peano arithmetic, which as an extension to the rules for predicate logic given in [Kel] consists of further 8 axioms and one extra rule of deductionas presented in Figure 4 below (warning: Peano rules do not normally include the $\uparrow$ function, but we include it here for reasons which will be clear later).

$$\frac{}{\forall n\ \neg(n = n + 1)} \qquad \frac{}{\forall m \forall n\ ((m + 1 = n + 1) \rightarrow (m = n))}$$

$$\frac{}{\forall n\ (n + 0 = n)} \qquad \frac{}{\forall m \forall n\ (m + (n + 1) = (m + n) + 1)}$$

$$\frac{}{\forall n\ (n * 0 = 0)} \qquad \frac{}{\forall m \forall n\ (m * (n + 1) = (m * n) + m)}$$

$$\frac{}{\forall n\ (n \uparrow 0 = 1)} \qquad \frac{}{\forall m \forall n\ (m \uparrow (n + 1) = (m \uparrow n) * m)}$$

$$\frac{\varphi(0) \quad \forall n\ (\varphi(n) \rightarrow \varphi(n + 1))}{\forall n\ \varphi(n)}$$

Fig. 4. Peano rules for $N_{\uparrow}$

Example 4. Look again at your proof using the Hoare rules for the program $C_{\uparrow}$ from Exercise 1. This proof included a number of proof obligations of the form $\vdash_N \varphi \rightarrow \psi$, which were left without formal proofs. You should be able to convince yourself, that these proof obligations can indeed be proven using a combination of the proof rules from[Kel] and the Peano rules in Figure 4. □

It should be clear that the axioms and the rule in Figure 4 are sound for the interpretation of natural numbers N, that is all its theorems are indeed true in the interpretation N. The eight axioms state simple facts about natural numbers, addition, multiplication, and exponentiation, and the final rule of deduction is just a formulation of the principle of induction in natural numbers. However, it is not so clear whether these rules are enough, i.e. will they provide a complete set of rules for N? We leave this question until the final section, and ask here a slightly different question: if we add these rules to the predicate logic rules from [Kel] and the Hoare rules above, will we get a complete proof system for Hoare triples?

The answer is negative, which will be proved in the following. As a matter of fact, we shall prove the much stronger result, that not only is this particular extension incomplete, but any other extension would fail as well! And what exactly do we mean by "any other extension"? For the purpose of making this precise, we formalize the notion of a proof system as follows:

Definition 3. *Given a logic with a syntactic set of formulae φ. A proof system for φ consists of an alphabet Σ in which to write proofs, and a set of rules, such that for any string $\pi \in \Sigma^*$ and φ it is decidable whether π is a proof of φ.* □

This definition is deliberately stated in a very liberal form. The only assumption we make is, that whatever notion of proof system you have in mind you must state some kind of rules, which make it possible to decide whether

or not a proposed proof of something is indeed a proof according to your rules. Clearly this assumption holds for our suggested extension of the predicate logic rules from [Kel] with the Peano rules above, but, more importantly, for *any* reasonable notion of what constitutes a proof system! With this definition we get the following immediate consequence.

Theorem 1. *For any proof system, the set of provable formulae (i.e. the set of theorems) is recursively enumerable.*

Proof. We need to argue for the existence of a Turing machine accepting the set of provable formulae. Consider the Turing machine which given a formula φ as input, lists the set of strings over Σ one by one, and for each of them decides whether it is a proof of φ (assumed to be decidable). The machine halts if and only if it reaches a proof of φ, i.e. if and only if φ is provable as required. □

Given these remarks, we will now prove the non-existence of *any* sound and complete proof system for PLN correctness by showing that the set of triples for which $\models_{par} \{\varphi\}C\{\psi\}$ is *not* recursively enumerable.

A proof of this statement could be given more or less directly from the existence of certain not recursively enumerable languages shown in [Mar], and some hand waiving arguments convincing you that Turing machines and PLN have "the same expressive power". However, we are going to give here a complete proof by a reduction from the complement of the Post's Correspondence Problem.

Before giving the proof let us recall the Post's Corrspondence Problem and argue that its complement is indeed not recursively enumerable.

Definition 4. *An instance of Post's Correspondence Problem (PCP) consists of two lists A and B, both of length $k > 0$, of strings over some alphabet Σ:*

$A = w_1, w_2, ..., w_k$
$B = x_1, x_2, ..., x_k$

A string of indices $i_0 i_1 ... i_{l-1} \in \{1, 2, ..., k\}^+$ is said to be a solution to A, B iff

$$w_{i_0} w_{i_1} ... w_{i_{l-1}} \ = \ x_{i_0} x_{i_1} ... x_{i_{l-1}}$$ □

Example 5. Consider the following PCP instance of lists A and B of length 2 :

A = ab, b
B = a, bb

In this case the string of indices 12 is a solution since

$w_1 w_2 = (ab)(b) = abb$ and
$x_1 x_2 = (a)(bb) = abb$

□

Theorem 2. *The set of no-instances to PCP , i.e. the set of instances without a solution, is not a recursively enumerable set.*

Proof. The set of yes-instances to PCP, i.e. the set of instances *with* a solution, is easily seen to be recursively enumerable (look for a solution by checking all possible index sequences one by one). Now, if the no-instances to PCP were also recursively enumerable, this would imply that both sets were recursive, according to [Mar] Theorem 10.5 (if L and L' are both recursively enumerable, then L is recursive). However, we know that this is not the case, since we have seen previously [Mar] a reduction from the unsolvable Accepting Problem for Turing machines, *Accepts*, to the set of yes-instances to PCP. □

So, we know that there exists no Turing machine (or any other realizable computational formalism), which when presented with a PCP instance, halts precisely if the instance does not have a solution. And hence, if we can reduce the set of PCP instances without any solutions to a language L, we can conclude that also L is not recursively enumerable. We are going to apply this technique to show that the language L consisting of triples $\models_{par} \{\varphi\}C\{\psi\}$ is not recursively enumerable. Recall that a reduction in this case means that we are going to provide a computable translation from PCP instances A, B to triples $\{\varphi_{A,B}\}C_{A,B}\{\psi_{A,B}\}$ such that A, B has no solution iff $\models_{par} \{\varphi_{A,B}\}C_{A,B}\{\psi_{A,B}\}$. Now PCP is a problem defined in terms of strings, and PLN is a language with natural numbers as its only data structure, and hence in order to provide our translation, we need somehow to look into ways of representing strings and their operations of composition etc. in terms of natural numbers and the operations of PLN. The purpose of the next section is to show exactly how this can and will be done.

5 Strings and Natural Numbers

We recall our standard way of bijectively representing natural numbers as decimal strings (base 10), and generalize this to an arbitrary base $b > 1$.

We are used to think of natural numbers as being represented in the decimal system, e.g. the string "124" as representing the natural number "$1 * 100 + 2 * 10 + 4 * 1$". This system is called decimal because 10 is used as "the base number" - the numbers $1, 2$ and 4 above we call the digits. However, this idea of representing natural numbers extends to any base number $b > 1$. Let us illustrate this more precisely. In doing so, we change our standard way of reading numbers by adopting the interpretation of "least significant digit first".

Example 6. With base number 3, the string "201" $\in \{0,1,2\}^*$ represents the number

$num_3(201) = 2 * 3^0 + 0 * 3^1 + 1 * 3^2 =$ the number 'eleven'

Correspondingly we say that the number 'eleven' is represented with base number 3 as the string 201, using the notation

$rep_3(\text{'eleven'}) = 201$ □

Definition 5. *Let b be any base number $b > 1$. $num_b : \{0,1,...,b-1\}^* \to N$ associates with any string $v = i_0 i_1 i_2 \dots i_{l-1} \in \{0,1,...,b-1\}^*$ the number*

$num_b(i_0 i_1 i_2 \dots i_{l-1}) = i_0 * b^0 + i_1 * b^1 + \dots i_{l-1} * b^{l-1}$ □

Hopefully, you see that num_b is just a natural generalization of our usual way of reading a sequence of decimals as a natural number. As we all know, reading decimal strings as numbers is not an injective mapping due to "dangling zeros" (in our normal way of writing numbers, 12 represents the same natural number as 012), but we are all accustomed to writing natural numbers *uniquely* as decimal strings without "dangling zeros". We shall be doing exactly the same in our generalization to any base in the following.

Definition 6. *Let b be any base number $b > 1$. Define N_b (our representations of natural numbers with base b) as the set of strings from $\{0,1,...,b-1\}^*$ of the form*

$N_b = \{\Lambda\} \cup \{0,1,...,b-1\}^*\{1,...,b-1\}$

Now define $rep_b : N \to N_b$ as follows:

$$rep_b(n) = \begin{cases} \Lambda & \text{if } n = 0 \\ rem(n,b) \bullet rep_b(div(n,b)) & \text{if } n > 0 \end{cases}$$

□

Exercise 2. What is the string rep_5('one hundred and thirty seven')? What natural number is $num_5(401)$? Can you find other strings v such that $num_5(v) = num_5(401)$? □

The next two Propositions state that num_b and rep_b are bijections between N and N_b.

Proposition 1. *Let b be any base number $b > 1$. Then for all natural numbers n,*

$num_b(rep_b(n) = n.$ □

Proof. Follows by a simple inductive argument in the length of $rep_b(n)$, simply applying the definitions of num_b and rep_b. □

Proposition 2. *Let b be a base number $b > 1$. For all $w \in N_b$*

$$rep_b(num_b(w)) = w.$$

Proof. Follows by a simple inductive argument in the length of w, simply applying the definitions of num_b and rep_b. □

Exercise 3. Show the two propositions above, and conclude that num_b and rep_b are *bijections*.

Finally, we need a few basic results showing how we may compute string operations on their num_b representations as natural numbers.

Proposition 3. *Let b be a base number $b > 1$. For any natural number n and any i such that $0 \leq i < |rep_b(n)|$. Then the unique i'th digit in $rep_b(n)$ as defined above is equal to $rem(div(n, b^i), b)$.*

Proof. Follows by a simple inductive argument in i. The base case $i = 0$ follows from definition of $rep_b(n)$. For the induction step, observe that the (i+1)st digit in $rep_b(n)$ from definition is equal to the i'th digit in $rep_b(div(n, b)$, which from induction hypothesis is equal to $rem(div(div(n, b), b^i), b) = rem(div(n, b^{i+1}), b)$. □

Note that we can conclude from Proposition 3 and Example 2, that the digits of $rep_b(n)$ can be computed in PLN!

Proposition 4. *Let b be a base number $b > 1$. For all $v, w \in N_b$*

$$num_b(vw) = num_b(v) + num_b(w) * b^{|v|}$$ □

Exercise 4. Show the Proposition above. □

Note that we can conclude from Proposition 4 and Example 1, that composition of strings from N_b can be computed in PLN using their numerical representations (num_b-representations)!

6 Incompleteness Theorem for Hoare triples

We show that the set of true Hoare triples over PLN is not recursively enumerable, and conclude our first main incompleteness result: that no sound and complete proof system for Hoare triples over PLN can exist.

We are now ready to prove our first main theorem:

Theorem 3. *There does not exist any sound and complete proof system for partial correctness of Hoare triples.*

Proof. The nonexistence of a sound and complete proof system clearly follows from Theorem 1 and the following statement:

The set of assertions $\models_{par} \{\varphi\}C\{\psi\}$ *is NOT recursively enumerable*

The proof is deliberately structured as a preparation for the proof of the main result of this note, Gödel's incompleteness theorem, to be stated and proved in the next section.

So, our reduction task is to construct for any PCP instance A, B a Hoare triple $\{\varphi_{A,B}\}C_{A,B}\{\psi_{A,B}\}$ such that $\models_{par} \{\varphi_{A,B}\}C_{A,B}\{\psi_{A,B}\}$ if and only if the given PCP instance has *no* solution. Our reduction will go as follows.

Given a PCP instance over some alphabet with lists Σ, $A = \{w_1, w_2, \ldots, w_k\}$ and $B = \{x_1, x_2, \ldots, x_k\}$, where the w_i's and the x_i's are finite, nonempty strings over Σ, we claim that we may construct from A and B a command $C_{A,B}$ such that $\models_{par} \{\top\}C_{A,B}\{\bot\}$ if and only if the PCP instance A, B has no solution. Notice that $\models_{par} \{\top\}C_{A,B}\{\bot\}$ is simply stating that the command $C_{A,B}$ when started in any initial state, if it terminates, then the final state satisfies $\bot$. But no state satisfies $\bot$ from definition of $\bot$, and hence $\models_{par} \{\top\}C_{A,B}\{\bot\}$ expresses that $C_{A,B}$ does not terminate for any initial state. So, our task is to construct $C_{A,B}$ such that $C_{A,B}$ halts for at least one of its possible initial states if and only if the PCP instance A, B has a solution.

First, let us observe that we may assume without loss of generality that $\Sigma = \{1, 2, \ldots, |\Sigma|\}$ (a PCP instance A, B over $\Sigma = \{\sigma_1, \sigma_2, ..\sigma_n\}$ has a solution iff the system obtained by renaming all occurrences of σ_i with i has a solution). With this observation, we define the constant $b = \max(|\Sigma|, k) + 1$, i.e. we choose b large enough that we can represent individual Σ symbols as well as individual string indices $\{1, 2, \ldots, k\}$ as numbers from $\{1, 2, \ldots, b-1\}$. More precisely, we can choose to represent any sequence of length $l > 0$, $i_0 i_1 \ldots i_{l-1} \in \{1, 2, \ldots, k\}^+ \subseteq N_b$ as $num_b(i_0 i_1 \ldots i_{l-1})$, and any string $v = \sigma_{j_0}\sigma_{j_1}...\sigma_{j_m} \in \Sigma^+$ where $j_0 j_1 ... j_m \in \{1, 2, \ldots, n\}^+ \subseteq N_b$ as $num_b(v)$. In particular, we are going to use the latter numerical representation of the w_i's and the x_i's from our given PCP.

The construction of $C_{A,B}$ is given below. The idea behind $C_{A,B}$ is the following. The behavior of $C_{A,B}$ depends only on the initial value of one program variable, in, which you may think of as its "input". Now, $C_{A,B}$ first computes in program variable j the number of significant digits in the b-ary representation of the initial value of in, $rep_b(\mathsf{in}) = i_0 i_1 \ldots i_{l-1}$. Next $C_{A,B}$ iteratively (in a while loop) computes in two program variables w and x values as follows. The first

pair of values satisfy $\mathsf{w} = num_b(w_{i_{l-1}})$ and $\mathsf{x} = num_b(x_{i_{l-1}})$, and the final values satisfy $\mathsf{w} = num_b(w_{i_0}w_{i_1}\ldots w_{i_{l-1}})$ and $\mathsf{x} = num_b(x_{i_0}x_{i_1}\ldots x_{i_{l-1}})$. Having computed these two numbers, $C_{A,B}$ will terminate only if $\mathsf{w} = \mathsf{x}$. Since we have chosen reprentations of the strings from A, B in the alphabet $\{1, 2, .., b-1\}$, we get from Corollary 1, that this is equivalent to $w_{i_0}w_{i_1}\ldots w_{i_{l-1}} = x_{i_0}x_{i_1}\ldots x_{i_{l-1}}$, i.e. $C_{A,B}$ will terminate precisely if $i_0 i_1 \ldots i_{l-1}$ is a solution to the given instance of PCP.

For ease of presentation we use the following PLN short hand notations:

"skip" stands for "$\mathsf{y} := \mathsf{y}$" (statement with no effect)
"loop" stands for "while true do skip" (non-terminating statement).

With this notation, $C_{A,B}$ is given in Figure 5. Please note that we only use primitives, which we have carefully argued can be computed in PLN. Also note that $C_{A,B}$ may be constructed algorithmically from A, B over alphabet Σ by computing the following constants used in the program (we use a bold type for these constants to distinguish these form program variables in $C_{A,B}$)

$\mathbf{k}$ = the number of strings in A (and B)
$\mathbf{b} = max(|\Sigma|, k) + 1$
$\mathbf{num_b(w_i)}$ for all $1 \leq i \leq k$
$|\mathbf{w_i}|$ for all $0 < i \leq k$
$\mathbf{num_b(x_i)}$ for all $1 \leq i \leq k$
$|\mathbf{x_i}|$ for all $0 < i \leq k$

We have annotated the program in Figure 5 following the notation used in [dADS]. The core of the proof of correctness of our reduction is to see that this annnotation is actually correct in the sense of [dADS]. The main point is to see that the annotated invariant Inv for the second while loop *is* an invariant. However, this follows more or less immediately form Propositions 3 and 4, stating how to compute the digits in rep_b representations of numbers (in) and string concatenation in terms of the num_b representations of strings (w and x).

Assume that the PCP instance A, B has a solution $i_0 i_1 \ldots i_{l-1} \in \{1, 2, ...k\}^+$. Let us look at the behaviour of $C_{A,B}$ when started in a state, in which the value of in is equal to $num_b(i_0 i_1 \ldots i_{l-1})$. Since $i_0 i_1 \ldots i_{l-1}$ is a string over $\{1, 2, ..k\} \subseteq N_b$, we clearly have $rep_b(\mathsf{in}) = rep_b(num_b(i_0 i_1 \ldots i_{l-1})) = i_0 i_1 \ldots i_{l-1}$ (from Proposition 2). According to the annotation of $C_{A,B}$, it will compute in w and x values such that $\mathsf{w} = num_b(w_{i_0}w_{i_1}\ldots w_{i_{l-1}})$ and $\mathsf{x} = num_b(x_{i_0}x_{i_1}\ldots x_{i_{l-1}})$. From our assumption that $i_0 i_1 \ldots i_{l-1}$ is a solution to A, B, we know that $w_{i_0}w_{i_1}\ldots w_{i_{j-1}} = x_{i_0}x_{i_1}\ldots x_{i_{l-1}}$, and hence the final values of w and x will be equal, and hence $C_{A,B}$ will terminate (see the final if statement of $C_{A,B}$).

So, if A, B has a solution, $C_{A,B}$ does not satisfy $\models_{par} \{\top\}C_{A,B}\{\bot\}$.

On the other hand, if A, B has no solution, we argue that $C_{A,B}$ does satisfy $\models_{par} \{\top\}C_{A,B}\{\bot\}$. Consider an arbitrary initial value of in, $\sigma(\mathsf{in})$, we need to ar-

$\{rep_b(\mathsf{in}) = i_0 i_1 \dots i_{l-1}\}$
if in = 0 then loop;
j := 1; while div(in, $\mathbf{b}^{\mathsf{j}}$) > 0 do j:= j+1;
$\{rep_b(\mathsf{in}) = i_0 i_1 \dots i_{l-1} \wedge \mathsf{j} = l > 0\}$
w := 0; x := 0;
while j > 0 do
  $\{Inv \equiv rep_b(\mathsf{in}) = i_0 i_1 \ldots i_{l-1} \wedge l > 0 \ \wedge i_j, .., i_{l-1} \in \{1, .., k\} \wedge$
    $\mathsf{w} = num_b(w_{i_j} \ldots w_{i_{l-1}}) \wedge \mathsf{x} = num_b(x_{i_j} \ldots x_{i_{j-1}})\}$
  j := j -1;
  i := rem(div(in, $\mathbf{b}^{\mathsf{j}}$), **b**);
  $\{rep_b(\mathsf{in}) = i_0 i_1 \ldots i_{l-1} \wedge l > 0 \wedge i_{j+1}, .., i_{l-1} \in \{1, .., k\} \wedge \mathsf{i} = i_j \wedge$
    $\mathsf{w} = num_b(w_{i_{j+1}} \ldots w_{i_{l-1}}) \wedge \mathsf{x} = num_b(x_{i_{j+1}} \ldots x_{i_{j-1}})\}$
  if i = 1 then w := $\mathbf{num_b(w_1)} + \mathsf{w} * \mathbf{b}^{|\mathbf{w_1}|}$; x := $\mathbf{num_b(x_1)} + \mathsf{x} * \mathbf{b}^{|\mathbf{x_1}|}$ else
  $\vdots$
  if i = **k** then w := $\mathbf{num_b(w_k)} + \mathsf{w} * \mathbf{b}^{|\mathbf{w_k}|}$; x := $\mathbf{num_b(x_k)} + \mathsf{x} * \mathbf{b}^{|\mathbf{x_k}|}$ else
  loop;
$\{rep_b(\mathsf{in}) = i_0 i_1 \ldots i_{l-1} \in \{1, .., k\}^+ \wedge \mathsf{w} = num_b(w_{i_0} \ldots w_{i_{l-1}}) \wedge \mathsf{x} = num_b(x_{i_0} \ldots x_{i_{l-1}})\}$
if w = x then skip else loop
$\{rep_b(\mathsf{in}) = i_0 i_1 \ldots i_{l-1} \in \{1, .., k\}^+ \wedge$
$\mathsf{w} = num_b(w_{i_0} \ldots w_{i_{j-1}}) \wedge \mathsf{x} = num_b(x_{i_0} \ldots x_{i_{j-1}}) \wedge \mathsf{w}{=}\mathsf{x}\}$

Fig. 5. PLN command $C_{A,B}$

gue that $C_{A,A}$ does not terminate. We spilt our argument in three cases depending on the value of $\sigma(\mathsf{in})$. If $\sigma(\mathsf{in}) = 0$, $C_{A,B}$ is constructed to loop (see first line of $C_{A,B}$). If $\sigma(\mathsf{in}) > 0$, the second while loop of $C_{A,B}$ computes (amongst other things) $rep_b(\sigma(\mathsf{in})) = i_0 i_1 \dots i_{l-1}$, and if $rep_b(\sigma(\mathsf{in}))$ is not a member of $\{1, .., k\}^+$ then $C_{A,B}$ is constructed to loop (see if-statement in the second while-statement). Finally, if $rep_b(\sigma(\mathsf{in})) = i_0 i_1 \dots i_{l-1} \in \{1, .., k\}$, $C_{A,B}$ will compute the values $\mathsf{w} = num_b(w_{i_0} w_{i_1} \dots w_{i_{l-1}})$ and $\mathsf{x} = num_b(x_{i_0} x_{i_1} \dots x_{i_{l-1}})$. Since A, B is assumed not to have a solution, we know that $w_{i_0} w_{i_1} \dots w_{i_{l-1}} \neq x_{i_0} x_{i_1} \dots x_{i_{l-1}}$. But now Proposition 2 implies that $\mathsf{w} \neq \mathsf{x}$, and hence $C_{A,B}$ is constructed to loop (in the final if-statement).

So, if A, B has no solution, $C_{A,B}$ satisfies $\models_{par} \{\top\} C_{A,B} \{\bot\}$.

Finally, we conclude the correctness of our reduction, that the constructed command $C_{A,B}$ is such that

A, B has no solution iff $C_{A,B}$ satisfies $\models_{par} \{\top\} C_{A,B} \{\bot\}$. □

Example 7. Consider the concrete instance of PCP over the alphabet $\Sigma = \{1, 2\}$, with lists $A_1 = \{12, 2\}$ and $B_1 = \{1, 22\}$. For this instance the number of strings in A_1 (and B_1) is $\mathbf{k} = 2$, and the base number **b** chosen in the reduction above will be 3 (why?). Furthermore, the required constants $\mathbf{num_b(w_i)}$, $|\mathbf{w_i}|$, $\mathbf{num_b(x_i)}$, and $|\mathbf{x_i}|$ are computed as follows:

$\mathbf{num_3}(\mathbf{w_1}) = num_3(12) = 7$
$|\mathbf{w_1}| = |12| = 2$
$\mathbf{num_3}(\mathbf{w_2}) = num_3(2) = 2$
$|\mathbf{w_2}| = |2| = 1$
$\mathbf{num_3}(\mathbf{x_1}) = num_3(1) = 1$
$|\mathbf{x_1}| = |1| = 1$
$\mathbf{num_3}(\mathbf{x_2}) = num_3(22) = 8$
$|\mathbf{x_2}| = |22| = 2$

Hence the complete PLN, C_{A_1,B_1} program constructed in the reduction from this particular PCP instance looks as follows:

```
if in = 0 then loop;
j := 1; while div(in, 3^j) > 0 do j:= j+1;
w := 0; x := 0;
while j > 0 do
  j := j -1;
  i := rem(div(in, 3^j), 3);
  if i = 1 then w := 7 + w * 3^2; x := 1 + x * 3^1 else
  if i = 2 then w := 2 + w * 3^1; x := 8 + x * 3^2 else
  loop;
if w = x then skip else loop
```
□

Exercise 5. What is the behaviour of the PLN program C_{A_1,B_1} from the Example above if the initial value of in is 9? What is the behaviour if the initial value of in is 4? Can you find an initial value of in, for which the resulting program terminates? Does the program satisfy the correctness formula $\models_{par} \{\top\} C_{A_1,B_1} \{\bot\}$?
□

Exercise 6. Consider the PCP instance over the alphabet $\Sigma = \{1, 2\}$, with lists $A_2 = \{112, 2\}$ and $B_2 = \{1, 222\}$. Construct the PLN program C_{A_2,B_2} constructed in the proof of Theorem 3. Does the program satisfy the correctness specification $\models_{par} \{\top\} C_{A_2,B_2} \{\bot\}$? □

Exercise 7. Argue for the correctness of the annotation of program $C_{A,B}$ in Figure 5. □

Exercise 8. Argue by a suitable reformulation of the proof above, that a similar incompleteness result also holds for Hoare triples under total correctness. □

7 Gödels Incompleteness Theorem

We show that the set of true formulae from $N_{\uparrow}$ is not recursively enumerable, and conclude our second main incompleteness result: that no sound and complete proof system for properties of natural numbers (expressed in $N_{\uparrow}$) can exist (Gödels Incompleteness Theorem).

Reading the statement of our incompleteness theorem for PLN Hoare triples, you may well wonder what is the essential and inherent difficulty in reasoning about PLN expressed so strongly in this theorem. Is it due to complications of the nature of "computations"? Well, the answer is no: one may prove that the rules for PLN triples as presented in Figure 2 are actually *relatively* sound and complete, in the sense that if they were supplemented by a sound and complete proof system for formulae of the form $N \models \varphi' \rightarrow \varphi$, then the combined system would actually be sound and complete! So, the problem is really in reasoning about natural numbers! And this fact is precisely the content of Gödel's incompleteness theorem, which is considered to be one of the major achievements of mathematics in the 20'th century. At the time, a number of leading mathematicians, including Hilbert, were engaged in a major project of formalizing "all of mathematics", i.e. looking for proof rules capturing the nature of various parts of mathematics. But Gödel put this project to an abrupt ending, by showing the impossibility of the mission: even the small fragment of mathematics dealing with simple arithmetic, can not be formalized by any set of proof system whatsoever!

We are going to state and prove Gödel's theorem for the vocabulary $N_{\uparrow}$ introduced earlier. Gödel originally proved his incompleteness theorem for an even simpler vocabulary ($N_{\uparrow}$ without the function $\uparrow$). The power and the profound implications of the result are of course not significantly reduced by our small extension of the vocabulary (see Concluding Remarks). More importantly, we shall be able to provide a relatively simple (hope you agree) proof building directly on top of our proof of the incompleteness theorem for PLN triples.

For ease of reading, we use in the following the more standard notation m^n for $m \uparrow n$.

Our incompleteness result implies, of course, that the sound Peano proof system we showed in the previous section cannot be complete, i.e. there must exist some formula from $N_{\uparrow}$, which is true in the interpretation of natural numbers, but which cannot be proved in the Peano proof system. In this particular case it can be shown that

$$\forall\, n\ ((\neg(n = 0)) \rightarrow \exists\, m\ (n = m + 1))$$

is such a formula. But what if we added this as an axiom? Well, Gödel's incompleteness theorem tells us, that there will still be some $N_{\uparrow}$ formula true for natural numbers, which cannot be proved also in this extended proof system.

And no matter how many axioms rules of deduction you add to the Peano system in this way, there will always be properties of the natural numbers, which you will not be able to prove in the resulting proof system.

Before we state and prove this fundamental theorem, we shall look a little into the expressive power of our logical language $N_\uparrow$. Remember that it is nothing but predicate logic with the vocabulary consisting of constant symbols 0, 1, function symbols $+$, $*$, and $\uparrow$, and only one predicate symbol $=$. However, even with this seemingly simple language we may express a number of more complicated predicates on natural numbers (which we shall find useful in our proof to follow). As examples:

$m \leq n$ can be expressed in $N_\uparrow$ as $\exists x\ (n = m + x)$, and hence
$m < n$ can be expressed in $N_\uparrow$ as $(m \leq n \wedge \neg(m = n))$.

For convenience, we are also going to use a couple of additional function symbols. As a first example, we argue that we may add the function $div(m, n)$ interpreted as integer division without increasing the expressive power of $N_\uparrow$. The only predicate in $N_\uparrow$ is equality $=$, and imagine that we write a predicate $t[div(m, n)] = t'$ where the first term t has an occurrence of $div(m, n)$ as indicated. We leave it to the reader to see that we may express the intended meaning of this in $N_\uparrow$ as

$$\exists d(\exists r(m = (n * d) + r \wedge (r < n))) \wedge t[d] = t'),$$

Similarly we may now use the function $rem(m, n)$ standing for the remainder of integer division of m by n in a predicate like $t[rem(m, n)] = t'$ as shorthand for

$$\exists r\ ((m = (n * div(m, n)) + r \wedge (r < n)) \wedge t[r] = t'.$$

Finally, we are going to use one more function, $sel_b(m, j, k)$, which needs a little explanation. Remember that all the b-ary representations of a natural number m are of the form $rep_b(m)0^*$, i.e. the unique string of significant digits, $rep_b(m)$, followed by an arbitrary number of 0's. Hence it makes sense to talk about *the* jth digit in b-ary representations of m as the jth digit in $rep_b(m)00...$. We have already seen how the jth digit of $rep_b(m)$ can be computed in PLN as the value of the expression $rem(div(m, b^j, b)$. Now, $sel_b(m, j, k)$ is just a slight generalization of this, supposed to denote the number represented by selecting the k digits starting from the jth digit in $rep_b(m)00...$ (numbering the digits starting with first digit as number 0).

As an example, consider $b = 2$, and $m = 51 = (110011)_2$. Then the 3 digits starting from position 2 in 110011 is 001, and hence the value of $sel_2(51, 2, 3)$ will be $num_2(001) = 4$.

Following the reasoning behind computing the digits of $rep_b(m)$ in PLN and our shorthand notation for div and rem above, we get

$sel_b(m, j, k)$ standing for
"the number represented by the k digits from j to $j + k - 1$ in $rep_b(m)00..$"
can be expressed in $N_\uparrow$ as $rem((div(m, b^j), b^k))$.

Exercise 9. Compute the following numbers:

$$sel_2(62, 1, 4), \quad sel_2(62, 4, 7), \quad sel_3(62, 1, 4).$$

Argue that $sel_b(m, j, k)$ can be expressed in $N_\uparrow$ as claimed above. □

With these observations we are ready for the main theorem.

Theorem 4. *There does not exist any sound and complete proof system for the interpretation $N_\uparrow$, i.e. there exists no proof system such that a formula φ is provable, $\vdash_{N_\uparrow} \varphi$, if and only if φ is semantically true for natural numbers, $N_\uparrow \models \varphi$.*

Proof. The proof follows exactly the same structure as the proof of our first incompleteness theorem, i.e. the essential part is to show that the set of formulae for which $N_\uparrow \models \varphi$ is not recursively enumerable. And the proof is again a reduction from the complement of Post's Correspondence Problem. So our task is to show that for any instance of PCP with lists A and B we may construct algorithmically a predicate logic formula $\varphi_{A,B}$ such that $N_\uparrow \models \varphi_{A,B}$ if and only if the PCP instance A, B has no solution.

The construction is going to follow very closely the idea behind the construction of our PLN program $C_{A,B}$ above. More precisely, we are going to construct a formulae $\varphi_{A,B}$ which asserts the nonexistence of a solution to the PCP instance A, B. Since our logic has negation, we start be constructing a formula $\psi_{A,B}$ asserting the existence of a solution $i_0, i_1, \ldots i_{l-1}$ to A, B.

The essential observation behind the construction of $\psi_{A,B}$ is, that we may express in our logic the existence of a number l and the existence of two natural numbers w and x, which represent in a formal sense the complete sequences of values assigned to the program variables w and x in the second while loop of $C_{A,B}$, that is for w (and similarly for x) the sequence

$$0, num_b(w_{i_{l-1}}), num_b(w_{i_{l-2}} w_{i_{l-1}}), \ldots, num_b(w_{i_0} \ldots w_{i_{l-1}}).$$

Notice that the sequence $w_{i_{l-1}}$, $w_{i_{l-2}} w_{i_{l-1}}, \ldots, w_{i_0} \ldots w_{i_{l-1}}$ is an increasing sequence of strings, and that all the strings have length less than or equal to $m = max * l$, where max is the maximum length of all the strings from A and B. Hence we choose to represent the complete sequence as the natural number represented by l blocks of digits, each block having the length m, and by appending 0's to each of the strings of digits shorter than m. In other words, the sequence

$0,\ w_{i_{l-1}}, w_{i_{l-2}}w_{i_{l-1}}, \ldots, w_{i_0}\ldots w_{i_{l-1}}$

will be represented by the number (to ease the reading of this number we have under braced the blocks):

$$\underbrace{0\ldots\ldots\ldots\ldots\ldots 0}_{m}\underbrace{w_{i_{l-1}}\overbrace{0\ldots\ldots\ldots 0}^{m-|w_{i_{l-1}}|}}_{m}\underbrace{w_{i_{l-2}}w_{i_{l-1}}\overbrace{0\ldots.0}^{m-|w_{i_{l-2}}w_{i_{l-1}}|}}_{m}\ldots\underbrace{w_{i_0}\ldots w_{i_{l-1}}\overbrace{0\ldots\ldots 0}^{m-|w_{i_0}\ldots w_{i_{l-1}}|}}_{m}$$

This idea should explain the overall structure of the following formula $\psi_{A,B}$ expressing the *existence* of a solution to the PCP instance A, B. Note that the formula only makes use of primitives, which we have carefully argued are expressible in $N_\uparrow$. As with the construction of $C_{A,B}$, also the formula $\psi_{A,B}$ is constructed from A, B based on initial computation of certain constants. In order to distinguish these constants from logical variables, we again use a bold type for these constants. To be specific, $\psi_{A,B}$ makes use of the following A, B constants:

b: the same base number as used in the construction of $C_{A,B}$
k: the number of strings in A (and B)
max: the length of the longest string occurring in A or B
$\mathbf{num_b}(\mathbf{w}_i), \mathbf{num_b}(\mathbf{x}_i), |\mathbf{w}_i|, |\mathbf{x}_i|$: the $num_\mathbf{b}$ representations and lengths of the individual A, B strings

Following these remarks, the formula $\psi_{A,B}$ will have the following structure:

$$\begin{aligned}\psi_{A,B} \equiv \exists\, l.\exists\, m.\ \ &(l \geq 1 \wedge m \geq l * \mathbf{max} \wedge \\ &\exists\, w, x.\ \ (FIRST(m, w, x) \wedge NEXT(l, m, w, x) \wedge LAST(l, m, w, x)))\end{aligned}$$

This formula asserts the existence of natural numbers l (think of this as the length of a solution to A, B), $m \geq l{*}\mathbf{max}$ (any such m will do), w and x (think of these as representing the complete sequence of values of program variables w and x in $C_{A,B}$), such that the the formulae $FIRST(m, w, x), NEXT(l, m, w, x)$ and $LAST(l, m, w, x)$ hold. We need to show that we may construct $N_\uparrow$ formulae $FIRST, NEXT$, and $LAST$, such that they together express that w and x represent the sequences of values taken by w and x in a *terminating* $C_{A,B}$ computation as explained above.

$FIRST$ is intended to express the fact that the first block of m digits in w and x are all 0's - the representations of the initial values (0) of w and x. This is easily achieved by:

$$FIRST(m, w, x) \equiv (sel_\mathbf{b}(w, 0, m) = 0 \wedge sel_\mathbf{b}(x, 0, m) = 0)$$

$NEXT(l, m, w, x)$ is intended to express that for all j between 0 and $l-1$, there exists an index i_j between 1 and $\mathbf{k}$, such that the correspondence between the jth and the $(j+1)$st block of m digits in $num_b(w)$ (and similarly in $num_b(x)$) represents the intended prefixing with the string w_{i_j} (x_{i_j}).

This may be achieved by a formula in $N_\uparrow$, which expresses the existence of an index i_j such that when selecting the appropriate blocks in $num_b(w)$: $sel_b(w, m*j, m, w')$ and $sel_b(w, m*(j+1), m, w'')$, checks that w'' is indeed the number represented by "prefixing $num_b(w')$ with $num_b(w_{i_j})$" (and similarly for $num_b(x)$). Formally, this is achieved by:

$$
\begin{array}{l}
NEXT(l,m,w,x) \equiv \\
\quad \forall\, j.\ (0 \le j \wedge j < l) \rightarrow \\
\quad\quad \exists\, i.\ (1 \le i \ \wedge\ i \le \mathbf{k}\ \wedge \\
\quad\quad\quad (i = 1 \rightarrow \\
\quad\quad\quad\quad (sel_{\mathbf{b}}(w, m*(j+1), m) = \mathbf{num_b}(\mathbf{w}_1) + sel_{\mathbf{b}}(w, m*j, m) * \mathbf{b}^{|\mathbf{w}_1|} \\
\quad\quad\quad\quad \wedge\ sel_{\mathbf{b}}(x, m*(j+1), m) = \mathbf{num_b}(\mathbf{x}_1) + sel_{\mathbf{b}}(x, m*j, m) * \mathbf{b}^{|\mathbf{x}_1|})) \wedge \\
\quad\quad\quad (i = 2 \rightarrow \\
\quad\quad\quad\quad (sel_{\mathbf{b}}(w, m*(j+1), m) = \mathbf{num_b}(\mathbf{w}_2) + sel_{\mathbf{b}}(w, m*j, m) * \mathbf{b}^{|\mathbf{w}_2|} \\
\quad\quad\quad\quad \wedge\ sel_{\mathbf{b}}(x, m*(j+1), m) = \mathbf{num_b}(\mathbf{x}_2) + sel_{\mathbf{b}}(x, m*j, m) * \mathbf{b}^{|\mathbf{x}_2|})) \wedge \\
\quad\quad\quad \vdots \\
\quad\quad\quad (i = \mathbf{k} \rightarrow \\
\quad\quad\quad\quad (sel_{\mathbf{b}}(w, m*(j+1), m) = \mathbf{num_b}(\mathbf{w}_k) + sel_{\mathbf{b}}(w, m*j, m) * \mathbf{b}^{|\mathbf{w}_k|} \\
\quad\quad\quad\quad \wedge\ sel_{\mathbf{b}}(x, m*(j+1), m) = \mathbf{num_b}(\mathbf{x}_k) + sel_{\mathbf{b}}(x, m*j, m) * \mathbf{b}^{|\mathbf{x}_k|}))
\end{array}
$$

Finally, $LAST$ is intended to express that the $(l+1)$st blocks of m digits of w and x are equal, i.e. that $w_0 \dots w_{l-1} = x_0 \dots x_{l-1}$. This is easily achieved by:

$$
\begin{array}{l}
LAST(l,m,w,x) \equiv \\
\quad (sel_{\mathbf{b}}(w, m*l, m) = sel_{\mathbf{b}}(x, m*l, m))
\end{array}
$$

Now, we finally define the formula $\varphi_{A,B}$ as the negation of $\psi_{A,B}$, i.e. $\varphi_{A,B} \equiv \neg\psi_{A,B}$. We need to argue that A, B has no solution iff $N_\uparrow \models \varphi_{A,B}$, which amounts to showing that A, B has a solution iff $N_\uparrow \models \psi_{A,B}$.

So assume that A, B has a solution $(i_0 i_1 \dots i_{n-1})$. We leave it to the reader to verify that $N_\uparrow \models \psi_{A,B}$ by choosing the following values for the existentially quantified variables in $\psi_{A,B}$:

$$
\begin{array}{l}
l = n \\
m = l * \mathbf{max}, \\
w = num_{\mathbf{b}}(0^m w_{i_{l-1}} 0^{m-|w_{i_{l-1}}|} \dots w_{i_0} \dots w_{i_{l-1}} 0^{m-|w_{i_0} \dots w_{i_{l-1}}|}), \text{ and} \\
x = num_{\mathbf{b}}(0^m x_{i_{l-1}} 0^{m-|x_{i_{l-1}}|} \dots x_{i_0} \dots x_{i_{l-1}} 0^{m-|x_{i_0} \dots x_{i_{l-1}}|}).
\end{array}
$$

On the other hand, if $N_\uparrow \models \psi_{A,B}$, then there exists a natural number l, some $m > l * max$, and w and x satisfying $FIRST$, $NEXT$, and $LAST$. However $FIRST$ and $NEXT$ imply (iteratively applying Proposition 4) that there must exist some $i_0, i_1, ...i_{l-1} \in \{1, .., \mathbf{k}\}^+$, such that $num_b(w_{i_0} \dots w_{i_{l-1}})$ is the number represented by the digits in the block of m digits from position $m*l$ in $rep_b(w)0^*$, *and* that $num_b(x_{i_0} \dots x_{i_{l-1}})$ is the number represented by the digits in the block of m digits from position $m*l$ in $rep_b(x)0^*$. But now $LAST$ implies that these

two numbers are equal, which (from Proposition 2) implies that $w_{i_0} \dots w_{i_{l-1}} = x_{i_0} \dots x_{i_{l-1}}$, and hence $i_0 i_1 \dots i_{l-1}$ is a solution to the PCP instance A, B.

So, we conclude A, B has a solution iff $N_\uparrow \models \psi_{A,B}$, or equivalently A, B has no solution iff $N_\uparrow \models \varphi_{A,B}$ ($\equiv \neg\psi_{A,B}$), i.e. the correctness of our reduction. □

Exercise 10. Construct the formula $\varphi_{A,B}$ for the concrete PCP instance A, B from Exercise 6. Does $\varphi_{A,B}$ satisfy $N_\uparrow \models \varphi_{A,B}$? □

8 Concluding Remarks

One small remark on the choice of $N_\uparrow$ as the logical language for our version of Gödel's incompleteness theorem. It can be shown that $\uparrow$ can be expressed in the vocabulary of the other symbols, N_*, and hence we get Gödel's original incompleteness result for N_* as a corollary. So, you may ask if N_* could be further reduced to get a version of Gödel's result even for a simpler logical language? As an example, it can be shown that $*$ cannot be expressed in N_+, the predicate logic formulae with constants 0 and 1, function symbol $+$, and predicate symbol $=$. As a matter of fact, this logic (usually called Pressburger arithmetic) *does* have a sound and complete proof system – it is even decidable for a given formula whether or not it is true in the interpretation of natural numbers (can be proved using a connection to regular languages).

In summary, we have shown in this small note two foundational incompleteness results.

The incompleteness result for PLN Hoare triples is telling us, that the process of proving the correctness of computer programs is provably a task which simply *cannot* be automized. It would be wonderful for the IT industry (and its programmers in particular) if such a tool existed, but as we now know, it cannot exist! The response to this insight from computer science is twofold (at least). First of all, tools are developed (theorem provers), which are used in semi-automatic program verification, in the sense that programs are proved correct interactively with the programmer (asking the programmer himself to prove certain required properties of natural numbers and other datastructures). Secondly, the inherent difficulty in reasoning about programs has been the leading motivation behind the school of "structured programming", which in a way can be seen as reversing the problem of correctness as we have addressed it here: rather that applying logic to the problem of proving the correctness of given programs, start with the logical specifications of what you want to do, and develop the program from this in such a way that you are assured of the correctness of the end result.

Gödel's incompleteness theorem is in its formulation one of the most important contributions to the understanding of foundation of mathematics. But we

claim, that it is perhaps even more important for computer science! The result is telling us something about the severe limitations of what you can do syntactically (say in proof systems) attempting to reason semantically (say about natural numbers). And in a way, *all* of computing is concerned with exacly this problem: performing semantic tasks in terms of syntactic computations. Hence Gödel's incompleteness theorem is really telling us something fundamental about the limitations of computing. Even though the result is stated concretely in terms of the complexity of natural numbers, of course the incompleteness "lifts" to almost any data structure of interest: lists, trees, graphs, etc.

However, there are execptions to this rather depressing conclusion. A surprisingly large part of the software industry is now occupied with the development of so-called embedded systems (mobile phones, TV-controllers, etc.), which although they are typically large and complex systems, they are actually finite state systems. And some of the techniques we know from the theory of finite automata have turned out to be the main ingredients in the development of modern tools for proving properties of embedded software (so-called model checkers). Some of these issues are touched upon in the DAIMI course dDist.

We have seen here complete formal proofs of our two incompleteness theorems. The implications of these results have been treated in a number of books aimed at the general public. If you are interested in supplementing dModLog with some of these more "popularized" discussions, you may consider e.g. [Har,Hof,Nør,Pen].

Some of these, [Har] in particular, also discusses the practical implications of more recent foundational results from computer science, classifying the solvable problems according to their difficulty with respect to resource requirements (e.g time and space). The main point here is that certain problems "solvable in theory" (i.e. by Turing machines) can be shown to "unsolvable in practice" because of their inherent resource requirements. This is one of the topics of the DAIMI course dSøg.

[Hof] discusses amongst other things the implications of our incompleteness results from an Artificial Intelligence perspective. In particular, symbolic computations are treated using programming language paradigms completely different from the imperative paradigm of PLN (and $JAVA$). Such paradigms are treated also in the DAIMI course dSprog.

References

[dADS] Mikkel Nygaard Hansen, Erik Meineche Schmidt: *Algorithms and Data Structures, Transition Systems*, DAIMI-FN 64, 2003 .

[Har] David Harel: *Computers Ltd.: What They Really Can't Do*, Oxford University Press, 2000.

[Mar] John Martin: *Introduction to Languages and the Theory of Computation*, McGraw-Hill, 2003.

[Hof] D.R. Hofstadter: *Gödel, Escher, Bach: An Eternal Golden Braid*, Penguin Books, 1979.
[Kel] John Kelly: *The Essence of Logic*, Prentice Hall, 1997.
[Nør] Tor Nørretranders: *Mærk Verden*, Gyldendal, 1991.
[Pen] R. Penrose: *The Emperor's New Mind*, Oxford University Press, 1990.