Verified Proof of P=NP in the Silence Theorem Prover Language

Matias Scharager

March 2020

1 Abstract

Godel's Incompleteness Theorem has for quite a long time made people aware that it is not possible to design a sound, complete and consistent language that encapsulates all the natural numbers. Programming languages that care about type theory in their construction usually try to devise theorems on things that seem like the natural numbers and exhibit some properties of them, but never fully encapsulate all of mathematics. There will always be a theorem left unproven in these languages.

However, the common approach these languages have is that they neglect completeness. Instead of this, the Silence programming language implemented in this paper goes beyond this idea and chooses to neglect soundness. This allows for a language capable of proving any mathematical statement. We then proceed to add in as many formalisms as necessary to express the $P = NP$ problem and prove it.

2 Booleans

Our booleans consist of 3 distinct terms each of type **Boool**. **want** represents the true case. **don't want** represents the false case. And of course, **might want** is indecisive and doesn't know which one to pick.

$$\frac{}{\Gamma \vdash \textbf{want} : \textbf{Boool}} \qquad \frac{}{\textbf{want val}} \qquad \frac{}{\Gamma \vdash \textbf{don't want} : \textbf{Boool}} \qquad \frac{}{\textbf{don't want val}} \qquad \frac{}{\Gamma \vdash \textbf{might want} : \textbf{Boool}}$$

$$\frac{}{\textbf{might want} \longrightarrow \textbf{might want}} \qquad \frac{B \longrightarrow B'}{\textbf{I } B\ a \textbf{ more than } b \longrightarrow \textbf{I } B'\ a \textbf{ more than } b} \qquad \frac{}{\textbf{I want } a \textbf{ more than } b \longrightarrow a}$$

$$\frac{}{\textbf{I don't want } a \textbf{ more than } b \longrightarrow b}$$

3 Natural Numbers

It is generally a good thing to have some notion of numbers in a programming language. The reason behind this is that people generally associate the word computation as being something done on numbers and so having numbers in a programming language is a must. The following is the statics and dynamics for an encoding of the natural numbers in Silence.

$$\frac{}{\Gamma \vdash \mathbf{7} : \textbf{Nat}} \qquad \frac{}{\mathbf{7} \textbf{ val}} \qquad \frac{\Gamma \vdash x : \textbf{Nat}}{\Gamma \vdash \textbf{succccc}(x) : \textbf{Nat}} \qquad \frac{x \textbf{ val}}{\textbf{succccc}(x) \textbf{ val}} \qquad \frac{\Gamma \vdash N : \textbf{Nat} \quad \Gamma \vdash a : A \quad \Gamma, x : A \vdash b : A}{\Gamma \vdash \textbf{recNat}(N; a; x.b) : A}$$

$$\frac{N \longrightarrow N'}{\textbf{recNat}(N; a; x.b) \longrightarrow \textbf{recNat}(N'; a; x.b)} \qquad \frac{}{\textbf{recNat}(\mathbf{7}; a; x.b) \longrightarrow a}$$

$$\frac{}{\textbf{recNat}(\textbf{succccc}(N); a; x.b) \longrightarrow [\textbf{recNat}(N; a; x.b)/x]b}$$

As such, we now have Heyting Arithmetic. Notice to avoid arguments as to whether the natural numbers begin at 0 or 1, we have opted to start at 7.

4 Functions

All functions are fun, so we didn't want to leave any of them out except we only really want to implement three of them. Silence has fixed point recursive functions and non-recursive functions. It has dependent type functions as well.

$$\frac{\Gamma, a : A \vdash b : B}{\Gamma \vdash \textbf{fn}\ a : A \Rightarrow b : A \to B} \qquad \frac{\Gamma \vdash f : A \to B \quad \Gamma \vdash x : A}{\Gamma \vdash f\ x : B} \qquad \frac{}{\textbf{fn}\ a : A \Rightarrow b\ \textbf{val}} \qquad \frac{f \longrightarrow f'}{f\ x \longrightarrow f'\ x} \qquad \frac{x \longrightarrow x'}{f\ x \longrightarrow f\ x'}$$

$$\frac{a\ \textbf{val}}{(\textbf{fn}\ x : A \Rightarrow b)\ a \longrightarrow [a/x]b} \qquad \frac{\Gamma, a : A, f : A \to B \vdash b : B}{\Gamma \vdash \textbf{fun}\ f\ (a : A) \Rightarrow b : A \to B}$$

$$\frac{}{\textbf{fun}\ f\ (a : A) \Rightarrow b \longrightarrow \textbf{fn}\ a : A \Rightarrow [(\textbf{fun}\ f\ (a : A) \Rightarrow b)/f]b} \qquad \frac{\Gamma, a : A \vdash b : B(a)}{\Gamma \vdash \Lambda a : A.b : \forall a : A.B(a)}$$

$$\frac{\Gamma \vdash f : \forall a : A.B \quad \Gamma \vdash x : A}{\Gamma \vdash f\ x : [x/a]} \qquad \frac{}{(\Lambda a : A.b)\ x \longrightarrow [x/a]b}$$

5 Tuples

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vdash (a, b) : A \wedge B} \qquad \frac{\Gamma \vdash M : A \wedge B}{\Gamma \vdash M \cdot \mathbf{l} : A} \qquad \frac{\Gamma \vdash M : A \wedge B}{\Gamma \vdash M \cdot \mathbf{r} : B} \qquad \frac{a\ \textbf{val} \quad b\ \textbf{val}}{(a, b)\ \textbf{val}} \qquad \frac{a \longrightarrow a'}{(a, b) \longrightarrow (a', b)} \qquad \frac{b \longrightarrow b'}{(a, b) \longrightarrow (a, b')}$$

$$\frac{M \longrightarrow M'}{M \cdot \mathbf{l} \longrightarrow M' \cdot \mathbf{l}} \qquad \frac{}{(a, b) \cdot \mathbf{l} \longrightarrow a} \qquad \frac{M \longrightarrow M'}{M \cdot \mathbf{r} \longrightarrow M' \cdot \mathbf{r}} \qquad \frac{}{(a, b) \cdot \mathbf{r} \longrightarrow b}$$

6 Sums

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \textbf{inl}(A \vee B, a) : A \vee B} \qquad \frac{a\ \textbf{val}}{\textbf{inl}(A \vee B, a)\ \textbf{val}} \qquad \frac{M \longrightarrow M'}{\textbf{inl}(A \vee B, M) \longrightarrow \textbf{inl}(A \vee B, M')} \qquad \frac{\Gamma \vdash b : B}{\Gamma \vdash \textbf{inr}(A \vee B, b) : A \vee B}$$

$$\frac{b\ \textbf{val}}{\textbf{inr}(A \vee B, b)\ \textbf{val}} \qquad \frac{M \longrightarrow M'}{\textbf{inr}(A \vee B, M) \longrightarrow \textbf{inr}(A \vee B, M')} \qquad \frac{\Gamma \vdash M : A \vee B \quad \Gamma, x : A \vdash a : C \quad \Gamma, x : B \vdash b : C}{\Gamma \vdash \textbf{case}(M; x.a; y.b) : C}$$

$$\frac{M \longrightarrow M'}{\textbf{case}(M; x.a; y.b) \longrightarrow \textbf{case}(M'; x.a; y.b)} \qquad \frac{}{\textbf{case}(\textbf{inl}(A \vee B, n); x.a; y.b) \longrightarrow [n/x]a}$$

$$\frac{}{\textbf{case}(\textbf{inr}(A \vee B, n); x.a; y.b) \longrightarrow [n/y]b}$$

7 Sigmas

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B[a/x]}{\Gamma \vdash \langle a, x.b\rangle : \exists x : A.B(x)} \qquad \frac{\Gamma \vdash M : \exists x : A.B(x)}{\Gamma \vdash M \cdot \mathbf{l} : A} \qquad \frac{\Gamma \vdash M : \exists x : A.B(x)}{\Gamma \vdash M \cdot \mathbf{r} : B[M \cdot \mathbf{l}/x]} \qquad \frac{a\ \textbf{val} \quad b\ \textbf{val}}{\langle a, x.b\rangle\ \textbf{val}} \qquad \frac{a \longrightarrow a'}{\langle a, x.b\rangle \longrightarrow \langle a', x.b\rangle}$$

$$\frac{b \longrightarrow b'}{\langle a, x.b\rangle \longrightarrow \langle a, x.b'\rangle} \qquad \frac{M \longrightarrow M'}{M \cdot \mathbf{l} \longrightarrow M' \cdot \mathbf{l}} \qquad \frac{}{\langle a, x.b\rangle \cdot \mathbf{l} \longrightarrow a} \qquad \frac{M \longrightarrow M'}{M \cdot \mathbf{r} \longrightarrow M' \cdot \mathbf{r}} \qquad \frac{}{\langle a, x.b\rangle \cdot \mathbf{r} \longrightarrow b[a/x]}$$

8 Unit and Ununit

This is the most significant achievement of this paper. While unit exists in most programming languages, ununit is a novel new idea that creates the unsoundess feature of Silence. Note that if tilt your head while viewing the typing judgement for ununit, it looks like a person with a halo on their head.

$$\frac{}{\Gamma \vdash () : \mathbf{1}} \qquad \frac{}{()\ \textbf{val}} \qquad \frac{}{\Gamma \vdash \langle\rangle : \mathbf{-1}} \qquad \frac{}{\langle\rangle\ \textbf{val}} \qquad \frac{\Gamma \vdash x : \mathbf{-1} \quad \Gamma \vdash y : \mathbf{1}}{\Gamma \vdash x\ y : A} \qquad \frac{x \longrightarrow x'}{x\ y \longrightarrow x'\ y} \qquad \frac{y \longrightarrow y'}{x\ y \longrightarrow x\ y'} \qquad \frac{}{\langle\rangle()\ \textbf{val}}$$

9 IO Behavior

True to it's name, Silence has no IO behavior. It can neither read from the input nor write to the output. We have purely functional behavior as there is also no imperative computations.

10 Progress and Preservation

This language is safe. By this we mean that there will be no undefined behavior. If we can derive a type for a program, then it is either a value or steps to something else, thus we have progress.

Proof. Progress

Trivial by induction on statics □

We also have that if a program steps to another program, then the type of the program is preserved, thus we have preservation

Proof. Preservation

Trivial by induction on dynamics □

11 Runtime of Programs

To help us reason about program execution, we will create a new relation between programs and allow for transitivity

$$\frac{A \longrightarrow B}{A \Longrightarrow B} \qquad\qquad \frac{A \Longrightarrow B \quad B \Longrightarrow C}{A \Longrightarrow C}$$

Consider the program $(\textbf{fun}\ f\ (x:\mathbf{1}) \Rightarrow f\ x)\ ()$. If we formally reason about it's execution, we can draw the following proofs.

Proof. $\mathcal{D}$

$$\dfrac{\dfrac{\dfrac{}{\textbf{fun}\ f\ (x:\mathbf{1}) \Rightarrow f\ x \longrightarrow \textbf{fn}\ x:\mathbf{1} \Rightarrow (\textbf{fun}\ f\ (x:\mathbf{1}) \Rightarrow f\ x)\ x}}{(\textbf{fun}\ f\ (x:\mathbf{1}) \Rightarrow f\ x)\ () \longrightarrow (\textbf{fn}\ x:\mathbf{1} \Rightarrow (\textbf{fun}\ f\ (x:\mathbf{1}) \Rightarrow f\ x)\ x)\ ()}}{(\textbf{fun}\ f\ (x:\mathbf{1}) \Rightarrow f\ x)\ () \Longrightarrow (\textbf{fn}\ x:\mathbf{1} \Rightarrow (\textbf{fun}\ f\ (x:\mathbf{1}) \Rightarrow f\ x)\ x)\ ()}$$

□

Proof. $\mathcal{E}$

$$\dfrac{\dfrac{}{(\textbf{fn}\ x:\mathbf{1} \Rightarrow (\textbf{fun}\ f\ (x:\mathbf{1}) \Rightarrow f\ x)\ x)\ () \longrightarrow (\textbf{fun}\ f\ (x:\mathbf{1}) \Rightarrow f\ x)\ ()}}{(\textbf{fn}\ x:\mathbf{1} \Rightarrow (\textbf{fun}\ f\ (x:\mathbf{1}) \Rightarrow f\ x)\ x)\ () \Longrightarrow (\textbf{fun}\ f\ (x:\mathbf{1}) \Rightarrow f\ x)\ ()}$$

□

Proof. Let A represent the expression $(\textbf{fun}\ f\ (x:\mathbf{1}) \Rightarrow f\ x)\ () \Longrightarrow (\textbf{fun}\ f\ (x:\mathbf{1}) \Rightarrow f\ x)\ ()$ which is what we want to prove. Thus we obtain the following proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{D}\quad\mathcal{E}}{A}\quad\dfrac{\mathcal{D}\quad\mathcal{E}}{A}}{A}\quad\dfrac{\mathcal{D}\quad\mathcal{E}}{A}}{A}\quad\dfrac{\mathcal{D}\quad\mathcal{E}}{A}}{A}\quad\dfrac{\mathcal{D}\quad\mathcal{E}}{A}}{A}\quad\dfrac{\mathcal{D}\quad\mathcal{E}}{A}}{A}\quad\dfrac{\mathcal{D}\quad\mathcal{E}}{A}}{A}\quad\dfrac{\dfrac{\mathcal{D}\quad\mathcal{E}}{A}\quad\dfrac{\dfrac{\mathcal{D}\quad\mathcal{E}}{A}\quad\dfrac{\dfrac{\mathcal{D}\quad\mathcal{E}}{A}\quad\dfrac{\dfrac{\mathcal{D}\quad\mathcal{E}}{A}\quad\dfrac{\dfrac{\mathcal{D}\quad\mathcal{E}}{A}\quad\dfrac{\dfrac{\mathcal{D}\quad\mathcal{E}}{A}\quad\dfrac{\mathcal{D}\quad\mathcal{E}}{A}}{A}}{A}}{A}}{A}}{A}}{A}}{A}$$

□

Since we have shown that we can reduce a term to itself in some number of steps, this proves that the programming language isn't strongly normalizing. A more clever proof will show that these are the only possible reductions we can take on this program through induction on dynamics and thus we have also proved the failure of weak normalization. Of course, all this is blatantly obvious from having adding in a fixed point operator to begin with.

However, there are some programs that do in fact always terminate. A good example of such program is () as it terminates in 0 steps. We would like to be able to analyze the complexity of such an algorithm, but there is an overwhelming amount of non-determinism in Silence. To address this, we will remove non-determinism by using a deterministic random number generator and applying rules based on the generated number. We can now define a special type family called P **zooms through** x which takes a function P and its input x and represents the fact that this algorithm ran in polynomial time on the input.

12 Conclusion of P=NP

The following type is representative in some way of the classical $P = NP$ problem. As the type states forall verifiers, if that verifier runs in polynomial time and there exists a valid solution that makes the verifier return true, we have a valid algorithm that runs in polynomial time and is able to find a solution.

$$\forall verifier : \mathbf{Nat} \to \mathbf{Boool}.\ ((\forall x : \mathbf{Nat}.\ verifier\ \mathbf{zooms\ through}\ x) \to (\exists x : \mathbf{Nat}.\ verifier\ x)$$

$$\to (\exists algo : () \to \mathbf{Nat}.\ (algo\ \mathbf{zooms\ through}\ () \land verifier(algo()))))$$

Notice that the following is a valid term of this type and thus is a proof of P=NP

$\Lambda Easy : \mathbf{Nat} \to \mathbf{Boool}.$

$\mathbf{fn}\ as : \forall x : \mathbf{Nat}.\ Easy\ \mathbf{zooms\ through}\ x \Rightarrow$

$\mathbf{fn}\ pie : \exists x : \mathbf{Nat}.\ Easy\ x \Rightarrow \langle\rangle()$

13 Future Goals

Much like most published programming languages, there currently exists no working implementation of Silence. The following is a proposed compilation translation into assembly

$$\frac{}{\cdot \vdash e : \tau \rightsquigarrow \mathbf{nop}}$$

Note that the translated program results in the same IO behavior as the original program, so we can very efficiently express the content of the language.