Incompleteness, Undecidability and Automated Proofs
(Invited Talk)

Cristian S. Calude[1(✉)] and Declan Thompson[1,2]

[1] Department of Computer Science, University of Auckland, Auckland, New Zealand
cristian@cs.auckland.ac.nz
[2] Department of Philosophy, Stanford University, Stanford, USA
http://www.cs.auckland.ac.nz/~cristian
http://www.stanford.edu/~declan

Abstract. Incompleteness and undecidability have been used for many years as arguments against automatising the practice of mathematics. The advent of powerful computers and proof-assistants – programs that assist the development of formal proofs by human-machine collaboration – has revived the interest in formal proofs and diminished considerably the value of these arguments.

In this paper we discuss some challenges proof-assistants face in handling undecidable problems – the very results cited above – using for illustrations the generic proof-assistant Isabelle.

1 Introduction

Gödel's incompleteness theorem (1931) and Turing's undecidability of the halting problem (1936) form the basis of a largely accepted thesis that mathematics cannot be relegated to computers. However, the impetuous development of powerful computers and versatile software led to the creation of *proof-assistants* – programs that assist the development of formal proofs by human-machine collaboration. This trend has revived the interest in formal proofs and diminished considerably the value of the above thesis for the working mathematician.

An impressive list of deep mathematical theorems have been formally proved including Gödel's incompleteness theorem (1986), the fundamental theorem of calculus (1996), the fundamental theorem of algebra (2000), the four colour theorem (2004), Jordan's curve theorem (2005) and the prime number theorem (2008). In 2014 Hales's Flyspeck project team formally validated in [7] Hales's proof [21] of the Kepler conjecture. Why did Hales' proof by exhaustion of the conjecture – involving the checking of many individual cases using complex computer calculations – published in the prestigious *Annals of Mathematics*, need "a validation"? Because the referees of the original paper had not been "100 % certain" that the paper was correct.

Hilbert's standard of proof is becoming practicable due to proof-assistants.

© Springer International Publishing AG 2016
V.P. Gerdt et al. (Eds.): CASC 2016, LNCS 9890, pp. 134–155, 2016.
DOI: 10.1007/978-3-319-45641-6_10

Are proof-assistants able to handle undecidable problems, the very results which have been used to argue the impossibility of doing mathematics with computers? In what follows we discuss some challenges proof-assistants face in handling two undecidable problems – termination and correctness – using for illustrations the generic proof-assistant Isabelle.

2 Truth and Provability

2.1 Incompleteness

In 1931 K. Gödel proved his celebrated incompleteness theorem which states that *no consistent system of axioms whose theorems can be effectively listed (e.g., by a computer program or algorithm) is capable of proving all true relations between natural numbers (arithmetic).* In such a system there are statements about the natural numbers that are true, but unprovable within the system – they are called *undecidable statements in the system.* For example, the *consistency* of such a system can be coded as a true property of natural numbers which the system cannot demonstrate, hence it is undecidable in the system. Furthermore, extending the system does not cure the problem. Examples of systems satisfying the properties of Gödel's incompleteness theorem are Peano arithmetic and Zermelo–Fraenkel set theory. One can interpret Gödel's theorem as saying that *no consistent system whose theorems can be effectively listed is capable of proving any mathematically true statement.*

2.2 Undecidability

Five years later A. Turing proved a computationally similar result, *the existence of (computationally) undecidable problems*, i.e. problems which have no algorithmic solution. To this goal Turing proposed a mathematical model of computability, based on what today we call a Turing machine (program), which is widely, but not unanimously, accepted as adequate (the Church-Turing thesis). The halting problem is the problem of determining in a finite time, from a description of an arbitrary Turing program and an input, whether the program will finish running or continue to run forever. Turing's theorem shows that *no Turing program can solve (correctly) every instance of the halting problem.* Our ubiquitous computers cannot do everything, a shock for some.

2.3 Incompleteness vs. Undecidability

Gödel's undecidable statements depend on the fixed formal system; Turing's undecidable problems depend on the adopted mathematical model of computation. They are both *relative.* There is a deep relation between these results coming from the impossibility of dealing algorithmically with some forms of infinity. In what follows we present a form of Gödel's incompleteness theorem by examining the halting problem. By $N(P, v)$ we mean that the Turing program P

will *never* halt on input v. For any particular program P and input v, $N(P, v)$ is a perfectly definite statement which is either true (in case P will never halt in the described situation) or false (in case P will eventually halt). When $N(P, v)$ is false, this fact can always be demonstrated by running P on v. No amount of computation will suffice to demonstrate the fact that $N(P, v)$ is true. We may still be able to prove that a particular $N(P, v)$ is true by a logical analysis of P's behaviour, but, because of the undecidability of the halting problem, no such automated method works correctly in *all* cases.

Suppose that certain strings of symbols (possibly paragraphs of a natural language (English, for example)) have been singled out – typically with the help of axioms and rules of inference – as proofs of particular statements of the form $N(P, v)$. Operationally, we assume that we have an *algorithmic syntactic test* that can determine whether an alleged proof Π that "$N(P, v)$ is true" is or is not correct. There are two natural requirements for the rules of proof:

Soundness: If there is a proof Π that $N(P, v)$ is true, then P will never halt on input v.

Completeness: If P will never halt on input v, then there is a proof Π that $N(P, v)$ is true.

Can we find a set of rules which is both sound and complete? The answer is *negative*. Suppose, by absurdity, we had found some "rules" of proof which are both sound and complete. Suppose that the proofs according to these "rules" are particular strings of symbols on some specific finite alphabet. Let $\Pi_1, \Pi_2, \Pi_3, \ldots$ be the quasi-lexicographic computable enumeration of all finite strings on this alphabet. This sequence includes all possible proofs, as well as a lot of other things (including a high percentage of total non-sense). But, hidden between the non-sense, we have all possible proofs. Next we show how we can use our "rules" to solve the halting problem – an impossibility. We are given a Turing program P and an input v and have to test whether or not P will eventually halt on v. To answer this question we run in parallel the following two computations:

(A) the computation of P on v,

(B) the computation consisting in generating the sequence $\Pi_1, \Pi_2, \Pi_3, \ldots$ of all possible proofs and using, as each Π_i is generated, the syntactic test to determine whether or not Π_i is a proof of $N(P, v)$.

The algorithm (A) and (B) stops when either (A) stops or in (B) a proof Π_i for $N(P, v)$ is validated as correct: in the first case the answer is "P stops on v" and in the second case the answer is "P does not stop on v".

First we prove that the algorithm cannot stop simultaneously on both (A) and (B). If the algorithm stops through (B) then a proof Π_i for $N(P, v)$ is found, so by soundness the computation (A) cannot stop; if, on the contrary, the computation (A) stops, then no valid proof for $N(P, v)$ can exist because, by soundness, then P will never halt on input v.

Second we prove that the algorithm stops either on (A) or on (B). Indeed, if the algorithm does not stop on (A), then P will never halt on input v, so by

completeness the algorithm will stop on (B). If the algorithm does not stop on (B), then there is no valid proof for $N(P, v)$, so again by completeness, P will stop on v, hence the algorithm will stop through (A).

Finally we prove the correctness of the algorithm. If P will eventually halt on v, then the computation (A) will eventually stop and the algorithm will give the correct answer: "P stops on v". If P never halts on v, (A) will be of no use: however, because of completeness, there will be a valid proof Π_i of $N(P, v)$ which will be eventually *discovered* by the computation (B). Having obtained this Π_i we will be sure (because of soundness) that P will indeed never halt.

Thus, we have described an algorithm which would solve the halting problem, a contradiction! The conclusion is that *no rules of proof can be both sound and complete:* there is a true statement $N(P, v)$ which has no proof Π. This statement is *undecidable* in the system: it cannot be proved nor disproved (being true, this would contradict soundness).

The above proof does not indicate any particular pair (P, v) for which the "rules" cannot prove that $N(P, v)$ is true! We only know that there will be a pair (P, v) for which $N(P, v)$ is true, but not provable from the "rules". There always are other sound rules which decide the "undecidable" statement: for example, adding the "undecidable" (true) statement as an axiom we get a larger system which trivially proves the original "undecidable" statement. No matter how we try to avoid an "undecidable" statement the new and more powerful rules will in turn have their own undecidable statements.

2.4 Hilbert's Programme and Hilbert's Axiom

In the late 19th century mathematics – shaken by the discoveries of paradoxes (for example, Russell's paradox) – entered into a foundational crisis (in German, Grundlagenkrise der Mathematik) which prompted the search for proper foundations in the early 20th century. Three schools of philosophy of mathematics were opposing each other: formalism, intuitionism and logicism. D. Hilbert, the main exponent of formalism, held that *mathematics is only a language and a series of games, but not an arbitrary game with arbitrary rules.* Hilbert's programme proposed to ground all mathematics on a finite, complete set of axioms, and provide a proof that these axioms were consistent. Hilbert's programme included a formalisation of mathematics using a precise formal language with the following properties: *completeness*, a proof that all true mathematical statements can be proved in the adopted formal system, and *consistency*, a proof – preferably involving finite mathematical objects only – that no contradiction can be obtained in the formal system. The consistency of more complicated systems, such as complex analysis, could be proven in terms of simpler systems and, ultimately, the consistency of the whole of mathematics would be reduced to that of arithmetic.

In his famous lecture entitled "Mathematical problems", presented to the International Congress of Mathematicians held in Paris in 1900, Hilbert expressed his deep conviction in the solvability of all mathematical problems (cited from [18, p. 11]):

> Is the axiom of solvability of every problem a peculiar characteristic of mathematical thought alone, or is it possibly a general law inherent in the nature of the mind, that all questions which it asks must be answerable?... This conviction of the solvability of every mathematical problem is a powerful incentive to the worker. We hear within us the perpetual call: There is the problem. Seek its solution. You can find it by pure reason, for in mathematics there is no *ignorabimus*.

Thirty years later, on 8 September 1930, in response to *Ignoramus et ignorabimus* ("We do not know, we shall not know"), the Latin maxim used as motto by the German physiologist Emil du Bois-Reymond [10] to emphasise the limits of understanding of nature, Hilbert concluded his retirement address to the Society of German Scientists and Physicians[1] – the same meeting where Gödel presented his completeness and incompleteness theorems – with his now famous words:

> We must not believe those, who today, with philosophical bearing and deliberative tone, prophesy the fall of culture and accept the *ignorabimus*. For us there is no *ignorabimus*, and in my opinion none whatever in natural science. In opposition to the foolish *ignorabimus* our slogan shall be: "We must know. We will know." (in German: *Wir müssen wissen. Wir werden wissen*[2]).

2.5 Objective vs. Subjective Mathematics

According to Gödel [19], see also [18], objective mathematics consists of *the body of those mathematical propositions which hold in an absolute sense, without any further hypothesis.*

A mathematical statement constitutes an *objective problem* if it is a candidate for objective mathematics, that is, if its truth or falsity is independent of any hypotheses and does not depend on where or how it can be demonstrated. Problems in arithmetic are objective problems in contrast with problems in axiomatic geometry, which depend on their provability in a specific axiomatic system.

Gödel's *subjective mathematics* is the body of all humanly demonstrable or knowable mathematically true statements, that is, the set of all propositions which the human mind can in principle prove in some well-defined system of axioms in which every axiom is recognised to belong to objective mathematics and every rule preserves objective mathematics.

Does objective mathematics coincide with subjective mathematics? Gödel's answer (1951, see [19]) based on his incompleteness theorem was: *Either ... the*

[1] Included in the short radio presentation, see [24].

[2] The words are engraved on Hilbert's tombstone in Göttingen. This is a triple irony: their use as an epitaph, the fact that the day before the talk, Hilbert's optimism was undermined by Gödel's presentation of the incompleteness theorem, whose exceptional significance was, with the exception of John von Neumann, completely missed by the audience.

human mind ... infinitely surpasses the powers of any finite machine, or else there exist absolutely unsolvable ... problems.

Working with a *constructive* interpretation of truth values "true", "false" and modal "can be known" Martin-Löf stated the following theorem [27]: *There are no propositions which can neither be known to be true nor be known to be false.*

For the non-constructive mathematician this means that no propositions can be effectively produced (i.e. by an algorithm) of which it can be shown that they can neither be proved constructively nor disproved constructively. There may be absolutely unsolvable problems, but one cannot effectively produce one for which one can show that it is unsolvable.

2.6 Hilbert's Programme After Incompleteness

In the standard interpretation, incompleteness shows that most of the goals of Hilbert's programme were impossible to achieve ... However, much of it can be and was salvaged by changing its goals slightly. With the following modifications some parts of Hilbert's programme have been successfully completed. Although it is not possible to formalise all mathematics, it is feasible to formalise essentially all the mathematics that "anyone uses". Zermelo–Fraenkel set theory combined with first-order logic gives a satisfactory and generally accepted formalism for essentially all current mathematics. Although it is not possible to prove completeness for systems at least as powerful as Peano arithmetic (if they have a computable set of axioms), it is feasible to prove completeness for many weaker but interesting systems, for example, first-order logic (Gödel's completeness theorem), Kleene algebras and the algebra of regular events and various logics used in computer science. Undecidability is a consequence of incomputability: there is no algorithm deciding the truth of statements in Peano arithmetic. Tarski's algorithm (see [32]) decides the truth of any statement in analytic geometry (more precisely, the theory of real closed fields is decidable). With the Cantor-Dedekind axiom, this algorithm can decide the truth of any statement in Euclidean geometry. Finally, Martin-Löf theorem cited in the previous section strongly limits the impact of absolutely unsolvable problems, if any exist, as one cannot effectively produce one for which one could show that it is unsolvable.

3 Can Computers Do Mathematics?

Mathematical proofs are essentially based on axiomatic-deductive reasoning. This view was repeatedly expressed by the most prominent mathematicians. For Bourbaki [11], *Depuis les Grecs, qui dit Mathématique, dit démonstration.*

A *formal proof*, written in a formal language consisting of certain strings of symbols from a fixed alphabet, satisfies Hilbert's criterion of mechanical testing:

> The rules should be so clear, that if somebody gives you what they claim is a proof, there is a mechanical procedure that will check whether the proof is correct or not, whether it obeys the rules or not.

By making sure that every step is correct, one can tell once and for all whether a proof is correct or not, i.e. whether a theorem has been proved. Hilbert's concept of formal proof is an ideal of rigour for mathematics which has important applications in mathematical logic (computability theory and proof theory), but for many years seemed to be irrelevant for the practice of mathematics which uses *informal* (pen-on-paper) *proofs*. Such a proof is a rigorous argument expressed in a mixture of natural language and formulae that is intended to convince a knowledgeable mathematician of the truth of a statement, the theorem. Routine logical inferences are omitted. "Folklore" results are used without proof. Depending on the area, arguments may rely on intuition. Informal proofs are the standard of presentation of mathematics in textbooks, journals, classrooms, and conferences. They are the product of a social process. In principle, an informal proof can be converted into a formal proof; however, this is rarely, almost never, done in practice. See more in [15].

In the last 30 years a new influence on the mathematical practice has started to become stronger and stronger: *the impact of software and technology*. Software tools – called interactive theorem provers or proof-assistants – aiding the development of formal proofs by human-machine collaboration have appeared and got better and better. They include an interactive proof editor with which a human can guide the search for, the checking of and the storing of formal proofs, using a computer.

As discussed in Sect. 1, an impressive list of deep mathematical theorems have been formally proved. The December 2008 issue of the *Notices of AMS* includes four important papers on formal proof. A formal proof in Isabelle for a sharper form of the Kraft-Chaitin theorem was given in [14]. In 2014 an automated proof of Gödel's ontological proof of God's existence was given in [9] and an automatic 13-gigabyte proof solved a special case of the Erdös discrepancy problem [26]; only a year later, Tao [31] gave a pen-and-paper general solution. The current longest automatic proof has almost 200-tb[3]: it solves the Boolean Pythagorean triples problem [23], a long-standing open problem in Ramsey theory. A compressed 68-gb certificate allows anyone to reconstruct the proof for checking.

Hilbert's standard of proof is practicable, it's becoming reality. However, as noted in [17],

> [T]he majority of mathematicians remain hesitant to use software to help develop, organize, and verify their proofs. Yet concerns linger over usability and the reliability of computerized proofs, although some see technological assistance as being vital to avoid problems caused by human error.

There are three main obstacles to a wider use of automated proofs [17]: (a) the lack of trust in the "machine", (b) the necessity of repeatedly developing foundational material, (c) the apparent loss of understanding in favour of the syntactical correctness (see also [13]). Current solutions involve (a) using

[3] The approximate equivalent of all the digitised texts held by the US Library of Congress.

a small "trusted" kernel on top of which employ a complicated software that parses the code, but ultimately calls the kernel to check the proof, or use an independent checker, (b) growing archives of formal proofs (see for example [2]) and developing more powerful automatic proof procedures, (c) developing environments in which users can write and check formal proofs as well as query them with reference to the symptoms of understanding [15,33] and write papers explaining formal proofs.

4 Formalised Computability Theory

Computability theory is an inherently interesting field for automated theorem proving. Since the limitations of computation being studied are true of the theorem provers themselves, formalised computability theory is like modifying the engine of a plane mid-flight.

Early work in formalised computability theory was completed by [30], who formalised the primitive recursive functions in the ALF (Another Logical Framework) proof-assistant [1], a predecessor of the contemporary Coq [3]. The purpose of this formalisation was to provide a computer-checked proof that the Ackermann function is not primitive recursive. As such, study into the relationship of the recursive functions to other forms of computation was not undertaken. A formalisation of Unlimited Register Machines (URM) was given in [36] (see also [35]), and it was shown that URMs can simulate partial recursive functions.[4] The converse was not shown however. The paper [36] also formalised partial recursive functions in Coq.

More recent work by Michael Norrish [28] has established a greater body of formal computability theory. Norrish formalises an implementation of the λ-calculus model of computation in the HOL4 system [4] and further defines the partial recursive functions and establishes the computational equivalence of these two models. A number of standard results are proven, including the existence of a universal machine (which is constructed), the identification of recursively enumerable sets with the ranges or domains of partially computable functions, the undecidability of the halting problem and Rice's theorem. For use in these results, Norrish implements the "dove-tailing" method, whereby a function is run on input $0, \ldots, n$ for n steps, and then $0, \ldots, n, n+1$ for $n+1$ steps, and so forth.

A formalisation of Turing machines is given in [8] in the Matita interactive theorem prover [6]. While the focus is on the use of Turing machines in complexity theory, a universal Turing machine is constructed, and its correctness proved.

An impressive formalisation of three models of computable functions can be found in [34]. Here, the authors define Turing machines, abacus machines

[4] Historically, the syntactic class of partial functions constructed recursively is called *partially recursive functions*, see [25,29]. This class coincides with the semantic partial functions implementable by standard models of computation (Turing machines, URMs, the λ-calculus etc.) – the *partially computable functions*.

and partial recursive functions in the Isabelle, and give a formal proof of the undecidability of the halting problem for Turing machines. Turing machines are shown to be able to model abacus machines, and abacus machines to model recursive functions. The bulk of [34]'s work is in creating a universal recursive function which takes encodings of Turing machines as inputs, and gives the same output as those machines would. This formally establishes not only the existence of universal functions, but also the equivalence of the three models of computation. While a universal Turing machine is not directly constructed, its existence can be inferred from the proofs that Turing machines can model abacus machines, which can in turn model recursive functions. This contrasts with the direct constructions of a universal Turing machine given in [8]. The establishment of the equivalence of recursive functions and the λ-calculus in [28] gives us formal proofs of the computational equivalence of the following four models: Turing machines, abacus machines, partial recursive functions and the λ-calculus.

4.1 Automated Proofs in Isabelle

Isabelle is a generic proof-assistant derived from the Higher Order Logic (HOL) theorem proving software, which in turn is a descendant of Logic for Computable Functions (LCF). LCF was developed in 1972, HOL became stable around 1988, and development of Isabelle started in the 1990s [20]. Isabelle is based on a small core set of logical principles from which theories can be built up. As such, the confidence with which we can claim any theorem proven in Isabelle to be true is the same confidence with which we can claim that the small core is true.

Isabelle provides a formal language to work in, and a set of proof methods, which allow it to prove statements using logical rules, definitions, and axioms, as well as already proved statements. Proofs in Isabelle are essentially natural deduction style. A structured proof language, Isar, is provided which aims to make proofs more human readable, and which serves to greatly reduce the learning curve required to use Isabelle. In addition to the standard proof methods, Isabelle has a feature called *sledgehammer*, which calls external automated theorem provers in an attempt to prove the current goal. Isabelle is developed jointly at the University of Cambridge, Technische Universität München and Université Paris-Sud [5].

An Isabelle proof proceeds in an interactive manner. The user makes a claim, and must then prove it. Isabelle's output (separate to the source code which the user writes) gives information like the goals currently needing to be proved and whether any redundancy in proofs has been detected. Using jEdit, the user can select a line of a completed proof, and check the output to see what was being done at that point – this "hook" into the proof allows for easier understanding of new proofs. Note that the Isabelle output is not included with the formal proof which the user ends up with. Since it is not exported to documentation, we have used an image of the output below.

4.2 Partial Recursive Functions in Isabelle

The formalisation of partial recursive functions in Isabelle given by Xu et al. in [34] makes use of a datatype `recf` of recursive functions. The constructors for this datatype follow standard conventions for partial recursive functions, [29]. Adapting the original Isabelle code definitions to more standard notation and using the notation $\boldsymbol{x}_{i;j}$ to mean $x_i, x_{i+1}, \ldots, x_j$ we have:

$$\begin{aligned}
z(\boldsymbol{x}) &= 0,\\
s(\boldsymbol{x}) &= x_0 + 1,\\
id_n^m(\boldsymbol{x}) &= x_n,\\
Cn^n(f,g)(\boldsymbol{x}) &= f(g_0(\boldsymbol{x}),\ldots,g_m(\boldsymbol{x})),\\
Pr^n(f,g)(\boldsymbol{x}) &= \begin{cases} f(\boldsymbol{x}_{0;n-2}), & \text{if } x_{n-1}=0,\\ g(\boldsymbol{x}_{0;n-2}, x_{n-1}-1, Pr^n(f,g)(\boldsymbol{x}_{0;n-2}, x_{n-1}-1)), & \text{otherwise,}\end{cases}\\
Mn^n(f)(\boldsymbol{x}) &= \mu\{y \mid f(\boldsymbol{x}_{0;n-1}, y) = 0\}.
\end{aligned} \tag{1}$$

Separately, the termination for partial recursive functions is defined in [34] as follows (notice that the clauses for z and s implicitly require a 1-ary list).

$$\begin{aligned}
&termi\ z([n])\\
&termi\ s([n])\\
(n < m \wedge |\boldsymbol{x}| = m) &\rightarrow termi\ id_n^m(\boldsymbol{x})\\
termi\ f(g_0(\boldsymbol{x}),\ldots,g_m(\boldsymbol{x})) \wedge \forall i\ termi\ g_i(\boldsymbol{x}) \wedge |\boldsymbol{x}| = n &\rightarrow termi\ Cn^n(f,g)(\boldsymbol{x})\\
\forall y < x_{n-1}\ termi\ g(\boldsymbol{x}_{0;n-2}, y, Pr^n(f,g)(\boldsymbol{x}_{0;n-2}, y)) \wedge&\\
termi\ f(\boldsymbol{x}_{0;n-2}) \wedge |\boldsymbol{x}| = n+1 &\rightarrow termi\ Pr^n(f,g)(\boldsymbol{x})\\
|\boldsymbol{x}| = n \wedge termi\ f(\boldsymbol{x}, r) \wedge f(\boldsymbol{x}, r) = 0 \wedge&\\
\forall i < r\ termi\ f(\boldsymbol{x}, i) \wedge f(\boldsymbol{x}, i) > 0 &\rightarrow termi\ Mn^n(f)(\boldsymbol{x})
\end{aligned}$$

To see how these definitions work, let us construct an implementation of the addition function + within the framework [34]. We will take this opportunity to demonstrate how proofs proceed in the Isabelle system (the rest of this section has been generated from Isabelle code).

Addition is fairly easy to define using primitive recursion. We simply follow the Robinson Arithmetic approach of defining

$$x + y = \begin{cases} x, & \text{if } y = 0,\\ (s(x)) + z, & \text{if } y = s(z).\end{cases}$$

Using primitive recursion, we have $+ := Pr^1(id_0^1, Cn^3(s, [id_2^3]))$. We note that we use indices starting at 0. This is very similar to the Isabelle source code:

```
definition "rec_add = (Pr 1 (id 1 0) (Cn 3 s [(id 3 2)]))"
```

Within Isabelle, we can prove that `rec_add` is indeed the addition function +. The following lemma achieves this through an induction on the second argument.

```
lemma [simp] : "rec_exec rec_add [m, n] = m + n"
apply(induction n)
by(simp_all add:rec_add_def)
```

First we have stated the statement of the lemma. The command `rec_exec` tells Isabelle to evaluate the function `rec_add` (that is, we are using Definition (1) from above). Note that since they are unbound, there is an implicit universal quantification over the variables m, n. We have flagged this lemma as a simplification (`[simp]`), which tells Isabelle's *simp* proof method that whenever it sees `rec_exec rec_add [m, n]` it can be replaced by `m+n`. Our first step in the proof is to apply induction to argument n. Isabelle determines that n is a natural number (since the recursive functions are defined over them) and so adopts the appropriate inductive hypothesis. As such there are two goals to prove. The first is that adding 0 to m returns m, and the second is that if addition is correct for $m + n$ it is also correct for $m + (n + 1)$. At this point, the Isabelle output gives the information shown in Fig. 1. Both subgoals can be proved easily by unpacking the definition of `rec_add`. Our final command is to apply the *simp* proof method to all remaining subgoals, making use of the definition of `rec_add`. The *simp* method utilises a large number of built in simplification rules, as well as those rules added to it by `[simp]` flags in an attempt to prove the current goal(s). Here it succeeds. Both commands *by* and *apply* apply the proof methods indicated; the difference is that *by* tells Isabelle we have finished the proof – it is a streamlining of an *apply* command followed by the *qed* end-of-proof command.

```
proof (prove): step 1

goal (2 subgoals):
 1. rec_exec rec_add [m, 0] = m + 0
 2. ⋀n. rec_exec rec_add [m, n] = m + n ⟹ rec_exec rec_add [m, Suc n] = m + Suc n
```

Fig. 1. The Isabelle output after applying the *induction* proof method.

Next, we will show that `rec_add` terminates on all inputs. This proceeds in a more complex fashion. First, we establish that the unpacked definition terminates (which requires a complete sub-proof), and then apply the definition to show that `rec_add` terminates.

```
lemma [simp] : "terminate rec_add [m, n]"
proof -
have "terminate (Pr 1 (id 1 0) (Cn 3 s [(id 3 2)])) ([m]@[n])"
  proof
  show "terminate (id 1 0) [m]" by (simp add: termi_id)
  show "length [m] = 1" by simp
  {fix y assume "y < n"
   have "terminate (Cn 3 s [id 3 2]) [m, y, rec_exec (Pr 1 (id 1 0) (Cn
3 s [id 3 2])) [m, y]]"
   proof
    show "length [m, y, rec_exec (Pr 1 (id 1 0) (Cn 3 s [id 3 2])) [m,
y]] = 3" by simp
    have "terminate (id 3 2) [m, y, rec_exec (Pr 1 (id 1 0) (Cn 3 s [id
3 2])) [m, y]]"
      by (simp add: termi_id)
    thus "∀g∈set [id 3 2]. terminate g [m, y, rec_exec (Pr 1 (id 1 0)
(Cn 3 s [id 3 2])) [m, y]]"
      by simp
    show "terminate s (map (λg. rec_exec g [m, y, rec_exec (Pr 1 (id 1
0) (Cn 3 s [id 3 2])) [m, y]]) [id 3 2])"
      by (simp add: termi_s)
   qed
  }
  hence "∀ y < n. terminate (Cn 3 s [(id 3 2)])
          ([m, y, rec_exec (Pr 1 (id 1 0) (Cn 3 s [(id 3 2)])) [m, y]])"
by blast
  thus "∀y<n. terminate (Cn 3 s [recf.id 3 2])
          ([m] @ [y, rec_exec (Pr 1 (recf.id 1 0) (Cn 3 s [recf.id 3
2])) ([m] @ [y])])" by simp
  qed
thus ?thesis by (simp add: rec_add_def)
qed
```

This proof uses a different style to the previous proof – the Isar mark up language. Isar is designed to reflect the style of informal proofs, and is intended to be fairly human readable. Commands such as *hence*, *show* and *thus* have strict meanings within the system, which are similar to their natural language meanings.

In the first step of the proof, we claim that $Pr^1(id_0^1, Cn^3(s, [id_2^3]))$ (our addition function) halts on the arbitrary inputs m, n. This is established inside the sub-proof. There we first show two simple facts – that the identity function terminates on `[m]` and that `[m]` is a list of length 1. Next, we must establish the following goal:

$$\forall y < n(terminate\ Cn^3(s, [id_2^3])([m, y, Pr^1(id_0^1, Cn^3(s, [id_2^3]))([m, y]))).$$

This establishes termination for the recursive cases of the addition function. We prove this statement by fixing an arbitrary $y < n$ and showing that it is true for that y. This requires another sub-proof, this time for the termination of

$Cn^3(s, [id_2^3])$. There are three goals to achieve: First that the correct number of arguments is supplied (i.e. the list is of length 3), second that id_2^3 terminates on a list of length 3 and third that the successor function terminates on a list of length 1. Each is a straightforward unpacking of definitions, achieved by *simp*.

Having established that $Pr^1(id_0^1, Cn^3(s, [id_2^3]))$ terminates on arbitrary inputs m, n, we show our *thesis* (namely, that `rec_add` terminates on arbitrary m, n) by applying the definition of `rec_add`. The command *qed* indicates the end of a (successful) proof.

5 Formalising the Halting Problem and Its Undecidability

In [34] a detailed mechanised proof of the undecidability of the halting problem, including proofs of correctness for all programs used, is given. The proof uses the Turing machine model of computation and follows, in broad strokes, the classical proof. The assumption is made of the existence of a Turing program H which can solve the halting problem. Specifically, given an encoding $\langle M, n\rangle$ of a Turing machine M and input n, H outputs 0 if M halts on n and 1 otherwise.

The following modification D of H is then constructed: $D\langle M\rangle = \infty$ if $M\langle M\rangle$ halts, and $D\langle M\rangle = 1$, otherwise, and a contradiction is reached by computing: $D\langle D\rangle = \infty$ iff $D\langle D\rangle$ halts.

Any formalisation requires a number of aspects in the proof to be made explicit. For example, the modification D must be constructed, and the changes to H shown to be correct. Furthermore, explicit notions of halting and correctness must be defined. Since proofs are computer checked, special care must be taken with the implementation of halting.

In what follows we give an overview of the formal proof provided in [34] of the undecidability of the halting problem. The formalisation of Turing machines uses a two-way infinite single tape Turing machine in which tape cells can be in one of two states – blank or occupied. The tape is represented by a pair (l, r) of lists, with l representing the cells to the left of the read/write head and r the cell being read and those to the right of it. Five actions are available; write blank, write occupied, move left, move right and do nothing. A Turing program is simply a list of pairs of actions and natural numbers representing states – the order of the pairs encodes which instruction maps to which state and input. An example program from [34] follows:

$$dither := [\overbrace{\overbrace{(W_{Bk}, 1)}^{\text{read } Bk}, \overbrace{(R, 2)}^{\text{read } Oc}}^{\text{state 1 (start)}}, \overbrace{(L, 1), (L, 0)}^{\text{state 2}}]$$

The program begins in state 1. If it reads a blank cell, it writes a blank cell and goes to state 1. If it reads an occupied cell, it moves right and goes to state 2. In state 2, it moves left, returning to state one if it saw a blank cell and going to state 0 (the halting state) if it saw an occupied cell. Hence this program halts

on a tape containing two occupied cells, and loops indefinitely on any tape with fewer such cells.

In order to ease construction of programs, a sequential composition of Turing programs is introduced. Essentially, this modifies the programs by increasing the state numbers of any subsequent programs and changing the halting state to the start state of each next program. This composition is used to combine three Turing programs: a copy program (to copy a machine's code so it can read it), the supposed program to solve the halting problem (H above) and a program to loop infinitely in certain cases (the "dither" program). This result is the machine D from above. *Both the copying program and dither must be proved correct.* We will outline how this proceeds for dither.

Correctness of a Turing program is established through the use of Hoare triples. Essentially, the triple $\{P\}p\{Q\}$ indicates that program p run on a tape satisfying P will result in a tape satisfying Q. We can also write $\{P\}p \uparrow$ to indicate that p run on a tape satisfying P will never halt. The program dither should satisfy the triples

$$\{\lambda tp.\exists k.tp = (Bk^k, \langle 1\rangle)\}\ dither\ \{\lambda tp.\exists k.tp = (Bk^k, \langle 1\rangle)\}$$
$$\{\lambda tp.\exists k.tp = (Bk^k, \langle 0\rangle)\}\ dither\ \uparrow$$

if it is to match the description above. The first statement can be established in Isabelle by calculation; provided a tape matching the first condition, run the dither program and see what happens. Due to the design of the implementation, this is straightforward and very easily automated. A proof of the second statement clearly cannot proceed in the same manner – *running dither on such a tape should result in an infinite loop*, a phenomenon which will affect any Isabelle simulation of the machine. Instead, the second statement can be established by an induction on the number of steps performed, starting with the given input tape.

Having established the correctness of copy and dither, one can proceed to prove the undecidability of the halting problem, following the standard method. Here a definition of the halting problem must be introduced. The property of a Turing machine p halting on an input n is defined using Hoare triples as follows:[5]

$$halts\ p\ n := \{\lambda tp.tp = ([], \langle n\rangle)\}\ p\ \{\lambda tp.\exists k, m, l.tp = (Bk^k, \langle m\rangle @ Bk^l)\}.$$

We then assume that a machine H exists which solves the halting problem. Formally within Isabelle this is captured by the following Hoare triples:

$$halts\ M\ n \rightarrow \{\lambda tp.tp = ([Bk], \langle(\langle M\rangle, n)\rangle)\}\ H\ \{\lambda tp.\exists k.tp = (Bk^k, \langle 0\rangle)\}$$
$$\neg halts\ M\ n \rightarrow \{\lambda tp.tp = ([Bk], \langle(\langle M\rangle, n)\rangle)\}\ H\ \{\lambda tp.\exists k.tp = (Bk^k, \langle 1\rangle)\}.$$

Then we define the diagonalising Turing machine *contra* by

$$contra := copy; H; dither$$

[5] In Isabelle, the @ symbol indicates concatenation of lists. Also note that this definition of *halts* assumes functions with some number of inputs and a single output.

where; indicates the sequential composition of the programs. The contradiction is now reached through reasoning established from the proofs of correctness for copy and dither, and the Hoare triple assumptions for H.

6 Correctness vs. Termination in Isabelle

In this section, we discuss some relations between the undecidable properties of correctness and termination of functions in Isabelle. As a formal proof-assistant, Isabelle is charged with being able to prove both these properties (or their negations) for arbitrary functions. A partially computable function is correct if it gives the expected (with respect to some specifications) output on every input. Correctness is a *relative notion* – a function may be syntactically fine, but if it was intended to do multiplication and actually does division, it is not correct. In contrast, the notion of termination of a function is *absolute*. The evaluation of a partially computable function f terminates on input x is equivalent to the mathematical property of f being defined on x. The two terminologies reflect the dual origins of computability theory: partially recursive functions are exactly the partially computable functions, i.e. the partial functions computed by Turing machines. Correctness is more undecidable than termination, see [16].

Proofs of correctness and termination form a critical part of computability theory and any formalisation of computability theory theorems cannot avoid such proofs. For this reason, any implementation of partial recursive functions within Isabelle needs to be able to handle correctness and termination. Clearly, as we have discussed above, this presents a challenge. All results discussed in this section have been generated from within the Isabelle system.

Suppose we have a unary partially computable function f (x and n are naturals) and define the following partial function:

$$g(x,n) = \begin{cases} f(x), & \text{if } n > 0, \\ 0, & \text{otherwise.} \end{cases}$$

Mathematically, for $n > 0$, $g(x,n)$ is defined if and only if $f(x)$ is defined. Since g is defined for all values of x when $n = 0$, we have $\mathrm{dom}(g) = \{(x,0) \mid x \in \mathbb{N}\} \cup \{(x,n+1) \mid x \in \mathrm{dom}(f) \wedge n \in \mathbb{N}\}$.

In practice, it is possible that n is the output of some other function which tests properties of x. For example, we might take $n = h(x)$. The resulting function $t(x) = g(x,h(x))$ has a domain which requires testing of h: $\mathrm{dom}(t) = \{x \in \mathbb{N} \mid x \in \mathrm{dom}(h) \wedge (h(x) > 0 \rightarrow x \in \mathrm{dom}(f))\}$.

If h is total and has the property that $h(x) > 0 \rightarrow x \in \mathrm{dom}(f)$ then this gives a computable restriction of f; in those cases where $f(x)$ is undefined (and possibly in some other cases), $g(x,h(x)) = 0$ since $h(x) = 0$. This can be useful for working with f without worrying about incomputability. Of course, it may be prudent to assume that $f(x) \neq 0$ for all x, so that we can identify when a potentially incomputable argument has been supplied.

Let us consider how g could be constructed in Isabelle, using the implementation of partial recursive functions from [34]. For any definition of g we should

be able (in principle) to establish three facts. First, that it meets the definition of g (and thereby is *correct*). Second, specifically that it returns 0 when $n = 0$. Third, that in the case $n = 0$ it terminates. This third requirement may seem superfluous at first – if the function returns 0 then surely it terminates – but as we will see soon, the definitions for recursive functions in this interpretation do not always result in termination behaving as expected.

Arguably the most obvious implementation of g is primitive recursion on n. Indeed, if we define g this way we are able to establish all three requirements in Isabelle with no difficulty. Instead, let us consider a function which we can prove "correct" in some sense, but which does not terminate.

Take `rec_times` to be a computable function for multiplication, and `rec_sg` to be the *signature* function: $signature(0) = 0$, $signature(x) = 1$ if $x > 0$.

These functions are defined as recursive functions using the implementation from [34]. The following lemmata show that `rec_times` and are correctly defined.

```
lemma "rec_exec rec_times [x,y] = x*y" using rec_times_def by simp
lemma "rec_exec rec_sg [x] = (if x > 0 then 1 else 0)" using rec_sg_def
by simp
```

Here we notice an interesting interplay between object and meta languages. The left-hand side of each lemma references a formally defined recursive function, using [34]'s implementation in Isabelle. For example, `rec_exec rec_times [x,y] = x*y` evaluates the formal function `rec_times` on input `[x,y]`. The right-hand side utilises built-in Isabelle functions, such as multiplication `*`, `if then` statements and greater-than `>`. These can be seen as meta-language operations, with `rec_times` and `rec_sg` in the object language. An interesting observation is that the Isabelle language is meta *with respect to these functions*. However, as part of a formal system, it would generally be regarded as the object language. The two lemmata show correctness of the functions, which follows from their definitions (suppressed for clarity) using the `simp` proof method.

Now consider the following function intended to implement g.

```
definition "g2 F = Cn 2 rec_times [Cn 2 F [id 2 0], Cn 2 rec_sg [id 2 1]]"
```

In this definition, we simply multiply $F(x)$ by $signature(n)$. If $n > 0$ then the answer will be $F(x)$ and if $n = 0$ the answer will be $F(x) \times 0$. In mathematical notation, we have

$$g_2(F) := Cn^2(\times, [Cn^2(F, [id_0^2]), Cn^2(signature, [id_1^2])]).$$

Expanding out the compositions, we have simply $g_2(F)(x, n) = F(x) \times signature(n)$. Indeed, we can establish this in Isabelle by unpacking the definitions.

```
lemma "rec_exec (g2 F) [x, n] = (rec_exec F [x])*(rec_exec rec_sg [n])"
by(simp add:g2_def)
```

The reader familiar with partial recursive functions, however, should have noted an important problem with g_2, if it is to implement g. *As defined,* $g(F)(x,0) = 0$, *regardless of whether or not* $F(x)$ *is defined.* However $F(x) \times 0 = 0$ only if $F(x)$ is defined. Specifically, for this construction we have $\mathrm{dom}(g_2) = \{(x,n) \mid x \in \mathrm{dom}(F)\}$, which differs from the domain for g. Hence g_2 does not implement g. Worryingly, we can still prove the following lemma in Isabelle.

```
lemma "rec_exec (g2 F) [x, n] = (if n>0 then rec_exec F [x] else 0)"
by (simp add: g2_def)
```

Once again this lemma follows by a simple unpacking of the definition. Notice that the `else` condition does not depend upon $F(x)$ being defined. According to this lemma, if $n = 0$ then $g_2(F)(x,0) = 0$ for arbitrary F, x. Indeed, we can be more specific.

```
lemma "rec_exec (g2 F) [x, 0] = 0"
  by(simp add:g2_def)
```

Does this then mean that this implementation of partial recursive functions in Isabelle is flawed? Arguably, yes. We have been able to show an incorrect lemma, or at least a lemma which is incorrect given the natural understanding of the `rec_exec` command. However we have not yet shown all the three facts needed to be established. And it is with termination that we (as might be expected) encounter problems.

```
lemma "terminate (g2 F) [x, 0]"
apply(simp add:g2_def)
proof
show "length [x, 0] = 2" by simp
show "terminate rec_times (map (λg. rec_exec g [x, 0]) [Cn 2 F [id 2 0],
Cn 2 rec_sg [id 2 (Suc 0)]])" by simp
show "∀g∈set [Cn 2 F [id 2 0], Cn 2 rec_sg [id 2 (Suc 0)]]. terminate g
[x, 0]"
  proof -
  have "terminate (Cn 2 rec_sg [id 2 (Suc 0)]) [x, 0]" using termi_id
termi_cn by simp
  moreover have "terminate (Cn 2 F [id 2 0]) [x, 0]"
    proof
    show "length [x, 0] = 2" by simp
    show "∀g∈set [id 2 0]. terminate g [x, 0]" using termi_id by simp
    show "terminate F (map (λg. rec_exec g [x, 0]) [id 2 0])" sorry
    qed
  ultimately show ?thesis by simp
  qed
oops
```

First we establish that the length of input is correct. We second show that `rec_times` terminates on the required inputs. Decoded, this second statement seems to be of the form *terminate* $F(x) \times$ *signature*(n). In fact it is slightly

more subtle. We are asked to show that `rec_times` terminates on $F(x)$ and $signature(0)$, but an inherent assumption in the Isabelle system is that these are both defined natural numbers; that the inputs are in a correct format. Since `rec_times` is a total function, we are able to prove this statement.

Finally, we are required to establish that both $Cn^2(F, [id_0^2])(x, 0)$ and $Cn^2(signature, [id_1^2])(x, 0)$ terminate. The second claim is simple; $Cn^2(signature, [id_1^2])(x, 0)$ is a primitive recursive function and so will terminate – an unpacking of definitions will establish this. The first claim is impossible to establish however. We have $Cn^2(F, [id_0^2](x, 0) = F(x)$ and so to establish that $Cn^2(F, [id_0^2](x, 0)$ terminates we must establish first that $F(x)$ terminates (that is, $F(x)$ is defined).

Since F is arbitrary, we cannot establish termination for g_2 on $(x, 0)$. Hence g_2 does not implement g correctly. The problem of incorrectly establishing the correctness of g_2 can then be explained by requiring that correctness should include termination. The implementation of partial recursive functions by [34] has split evaluation of functions from their termination as a way to overcome termination issues within Isabelle. This has come at the cost of clarity in the implementation, and a departure from the standard definition of partial recursive functions, where termination and evaluation are inextricably linked.

6.1 Reuniting Evaluation and Termination

In standard definitions of partial recursive functions, evaluation of the function is explicitly linked to the termination of any functions involved. For example, in [12] the first mention of each function type specifies its domain. Functions are built up recursively, and, for example, a function θ obtained through composition $\theta(x_1, \ldots, x_n) = \psi(\phi_1(x_1, \ldots, x_n), \ldots, \phi_m(x_1, \ldots, x_m))$ has the domain defined as $\text{dom}(\theta) = \{(x_1, \ldots, x_n) \in \mathbb{N}^n \mid (x_1, \ldots, x_n) \in \bigcap_{i=1}^{m} \text{dom}(\phi_i)$ and $(\phi_1(x_1, \ldots, x_n), \ldots, \phi_m(x_1, \ldots, x_n)) \in \text{dom}(\psi)\}$.

This explicit mentioning of domain contrasts with [34]'s implementation. In their implementation, the function executions are defined as having $\mathbb{N}^* = \mathbb{N} \cup \mathbb{N}^2 \cup \mathbb{N}^3 \cup \ldots$ as their domain. That is, Isabelle will happily (attempt to) evaluate a function on any input, regardless of whether it is in the domain of that function. Restrictions to domain, and to correct arity of arguments, are implemented entirely within the termination definitions.[6]

Partial recursive functions have domains built into them directly. Divorcing domains from the function definitions – motivated by the wish to increase understandability – is not a correct solution. Creating an implementation of partial recursive functions in Isabelle in which termination and evaluation are presented at once would be difficult. Isabelle requires proofs of termination for certain functions, which is likely part of the reason [34] decided to split the definitions. Proofs involving combined definitions are likely to be much messier

[6] Of course it should be noted that if an input is not within that domain of a function, Isabelle's attempt to evaluate is likely not to terminate. However, consequential strange behaviours can be observed, such as in g_2.

than the current model, since domains must explicitly be dealt with. From a formal perspective, the separation allows for separate proofs, which are more easily digested. However, it would be enough for a combined model to establish equivalence with the [34] model. If we could implement partial recursive functions in Isabelle, using a model defined as closely to a standard pen-and-paper definition (such as that provided by [12]) as possible, we would have greater confidence that model adequately represents the mathematical notion of partial recursive function. Subsequently establishing the equivalence of this model with that in [34] would allow the "importing" of results proved by [34]. This solution would mean greater ease of formal proofs from the split model while maintaining connection to the original model of partial recursive functions.

A tempting diversion in implementing partial recursive functions is ensuring they can be evaluated by the proof system. This would mean the proof system acting as an interpreter, and actually running the programs specified by the functions. When we evaluate `rec_add`, actual recursive calls are made to find the result.

This would be a very interesting approach – enlisting a modern computer to simulate a decades old model of computation. However, from a standpoint of formal proof, it is unnecessary. Proofs involving partial recursive functions at most require unpacking general definitions – explicit evaluation of functions is rarely required. That is, while evaluating addition through recursive calls may be fun, it is highly unlikely that any proofs will require it; since proofs generally deal with the abstract, we are less concerned with what $1 + 2$ is and more with how $x + y$ works. Due to this, it would be acceptable for an implementation of partial recursive functions to combine "evaluation" and termination at the expense of the system actually being able to evaluate the functions.

7 'Symptoms' of Undecidability in Isabelle

The problems in [34]'s implementation give one example of how undecidability impacts Isabelle. The careful nature in which the model is constructed, and the split of evaluation from termination are direct consequences of the undecidability of the halting problem. The *sledgehammer* feature, which searches for proofs to given claims, has a time restriction built in, again to combat undecidability.

Isabelle is a programming language, so its programs may terminate or not. When dealing with models of computation, what happens when Isabelle attempts to simulate such programs?

Isabelle has the ability to evaluate functions within the system. The user can type `value "1+2"` and Isabelle's output will display the answer. For basic functions this acts as a calculator, and for functions defined in Isabelle it can be used to ensure they behave as expected. It is interesting to see how Isabelle handles non-terminating computations, since identifying them is undecidable.

Consider the partial recursive function defined by $Mn^1(+)(x) = \mu\{y \mid x+y = 0\}$. It is obvious that $Mn^1(+)(0) = 0$ and $Mn^1(+)(x) = \infty$ for $x > 0$.

How then will Isabelle evaluate `value "rec_exec (Mn 1 rec_add) [1]"`? In fact Isabelle *refuses to try*, throwing instead a well-sortedness error. This makes

sense. The `Mn` function requires finding the least element of a possibly empty set. Since Isabelle has no guarantee the set is non-empty, it refuses to evaluate.

Isabelle's cautious nature comes at a cost, failing to evaluate *any* recursive function. The addition function is a recursive function and its definition does not use minimisation. This puts it into the class of computable functions. If Isabelle were to attempt to calculate `value "rec_exec rec_add [1, 1]"`, it would succeed. However, Isabelle again *refuses to try*, throwing the same well-sortedness error. Even a proof of general termination for `rec_add` does not help. Isabelle notices that the `rec_exec` definition incorporates a minimisation clause, and so refuses to have anything to do with evaluation. It even refuses to attempt evaluation of `value "rec_exec z [0]"`.

Yet all is not lost. We can still prove `rec_exec rec_add [1, 1] = 2`. In fact, this is almost trivial, since we have proved already the general statement that `rec_exec rec_add [m, n] = m + n`, and $m+n$ (the Isabelle function) can be calculated by Isabelle. This leads to an interesting contrast: *Isabelle will not attempt to evaluate partial recursive functions, but is happy to attempt to prove a claim made by the user.* While both operations involve skirting close to undefined functions, in proofs Isabelle can offload much responsibility to the user. For our original addition minimisation function $Mn^1(+)$, we can obviously not prove an output for any input other than 0, but `rec_exec (Mn 1 rec_add) [0] = 0` is provable in Isabelle.

8 Concluding Remarks

The nature of computability makes automated proofs a particularly interesting area. The landmark results of Gödel and Turing still loom large. Where they were discussing hypothetical computation models, we are using equivalent models to prove their own limitations. Yet we are still able to carve out larger sections of what can be achieved. To avoid the inherent complications, novel approaches need to be adopted, and the original proofs modified to achieve the required goals, given the abilities of modern proof-assistants (see, for example, [22]).

The formalisation of recursive functions we have considered is a good example of this. The model, though very similar, is not the traditional partial recursive functions as termination and evaluation have been separated. It would be nice if a combined model of partial recursive functions could be shown, within Isabelle, to be equivalent to the model provided in [34].

Great progress has been made in both formal proving and developing computability theory. Unexpectedly, the use of proof-assistants brings new connections between incompleteness and undecidability into sharp focus, so formal proving can contribute to semantics too. We expect that the role of proof-assistants for the working mathematician will steadily increase.

References

1. Alf homepage. http://homepages.inf.ed.ac.uk/wadler/realworld/alf.html. Accessed 25 Oct 2014
2. Archive of formal proofs. http://afp.sourceforge.net. Accessed 18 May 2016
3. Coq homepage. http://coq.inria.fr/. Accessed 25 Oct 2014
4. HOL4 homepage. http://hol.sourceforge.net/. Accessed 25 Oct 2014
5. Isabelle homepage. http://isabelle.in.tum.de/. Accessed 20 Oct 2014
6. Matita hompage. http://matita.cs.unibo.it/. Accessed 25 Oct 2014
7. Flyspeck project, September 2014. http://aperiodical.com/2014/09/the-flyspeck-project-is-complete-we-know-how-to-stack-balls
8. Asperti, A., Ricciotti, W.: Formalizing turing machines. In: Ong, L., de Queiroz, R. (eds.) WoLLIC 2012. LNCS, vol. 7456, pp. 1–25. Springer, Heidelberg (2012)
9. Benzmüller, C., Woltzenlogel Paleo, B.: Automating Gödel's ontological proof of God's existence with higher-order automated theorem provers. In: Schaub, T., Friedrich, G., O'Sullivan, B. (eds.) ECAI 2014, Frontiers in Artificial Intelligence and Applications, vol. 263, pp. 93–98. IOS Press (2014)
10. Du Bois-Reymond, E.H.: Über die Grenzen des Naturerkennens; Die sieben Welträthsel, zwei Vorträge. Von Veit, Leipzig (1898)
11. Bourbaki, N.: Theory of Sets. Elements of Mathematics. Springer, Heidelberg (1968)
12. Calude, C.: Theories of Computational Complexity, North-Holland, Amsterdam (1988)
13. Calude, C.S., Calude, E., Marcus, S.: Passages of proof. Bull. Eur. Assoc. Theor. Comput. Sci. **84**, 167–188 (2004)
14. Calude, C.S., Hay, N.J.: Every computably enumerable random real is provably computably enumerable random. Logic J. IGPL **17**, 325–350 (2009)
15. Calude, C.S., Müller, C.: Formal proof: reconciling correctness and understanding. In: Carette, J., Dixon, L., Coen, C.S., Watt, S.M. (eds.) MKM 2009, Held as Part of CICM 2009. LNCS, vol. 5625, pp. 217–232. Springer, Heidelberg (2009)
16. Cooper, S.B.: Computability Theory. Chapman Hall/CRC, London (2004)
17. Edwards, C.: Automated proofs. Math struggles with the usability of formal proofs. Commun. ACM **59**(4), 13–15 (2016)
18. Feferman, S.: Are there absolutely unsolvable problems? Gödel's dichotomy. Philosophia Math. **14**(2), 134–152 (2006)
19. Gödel, K.: Some basic theorems on the foundations of mathematics and their implications. In: Feferman, S., Dawson Jr., J.W., Goldfarb, W., Parsons, C., Solovay, R.M. (eds.) Collected Works. Unpublished Essays and Lectures. vol. III, pp. 304–323. Oxford University Press (1995)
20. Gordon, M.: From LCF to HOL: a short history. In: Proof, Language, and Interaction, pp. 169–186 (2000)
21. Hales, T.C.: A proof of the Kepler conjecture. Ann. Math. **162**(3), 1065–1185 (2005)
22. Hernández-Orozco, S., Hernández-Quiroz, F., Zenil, H., Sieg, W.: Rare speed-up in automatic theorem proving reveals tradeoff between computational time and information value (2015). http://arxiv.org/abs/1506.04349
23. Heule, M.J.H., Kullmann, O., Marek, V.W.: Solving and verifying the boolean Pythagorean triples problem via cube-and-conquer (2016). http://arxiv.org/abs/1605.00723v1 [cs.DM]

24. Hilbert, D.: Hilbert's 1930 radio speech. https://www.youtube.com/watch?v=EbgAu_X2mm4
25. Kleene, S.C.: Introduction to Metamathematics. North-Holland, Amsterdam (1952)
26. Konev, B., Lisitsa, A.: A SAT attack on the Erdös discrepancy conjecture (2014). http://arxiv.org/abs/1402.2184v2
27. Martin-Löf, P.: Verification then and now. In: De Pauli-Schimanovich, W., Koehler, E., Stadler, F. (eds.) The Foundational Debate, Complexity and Constructivity in Mathematics and Physics, pp. 187–196. Kluwer, Dordrecht (1995)
28. Norrish, M.: Mechanised computability theory. In: van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) ITP 2011. LNCS, vol. 6898, pp. 297–311. Springer, Heidelberg (2011)
29. Soare, R.I.: Recursively Enumerable Sets and Degrees: A Study of Computable Functions and Computably Generated Sets. Springer, Heidelberg (1987)
30. Szasz, N.: A machine checked proof that Ackermann's function is not primitive recursive. In: Huet, G. (ed.) Logical Environments, pp. 31–7. University Press (1991)
31. Tao, T.: The Erdös discrepancy problem (2015). http://arxiv.org/abs/1509.05363v5
32. Tarski, A.: A Decision Method for Elementary Algebra and Geometry. University of California Press, Berkeley and Los Angeles (1951)
33. Thompson, D.: Formalisation vs. understanding. In: Calude, C.S., Dinneen, M.J. (eds.) UCNC 2015. LNCS, vol. 9252, pp. 290–300. Springer, Heidelberg (2015)
34. Xu, J., Zhang, X., Urban, C.: Mechanising turing machines and computability theory in Isabelle/HOL. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) ITP 2013. LNCS, vol. 7998, pp. 147–162. Springer, Heidelberg (2013)
35. Zammit, V.: A mechanisation of computability theory in HOL. In: von Wright, J., Harrison, J., Grundy, J. (eds.) TPHOLs 1996. LNCS, vol. 1125. Springer, Heidelberg (1996)
36. Zammit, V.: On the Readability of Machine Checkable Formal Proofs. Ph.D. Thesis, University of Kent, March 1999