Computability in Constructive Type Theory

A dissertation submitted towards the degree of Doctor of Natural Sciences (Dr. rer. nat.)
of the Faculty of Mathematics and Computer Science of Saarland University

Yannick Forster

Saarbrücken, 2021

Dean:	Prof. Dr. Thomas Schuster
Thesis Defense:	November 26th, 2021
Advisor:	Prof. Dr. Gert Smolka
Examination Board:	Prof. Bernd Finkbeiner, PhD (Chair)
	Prof. Dr. Gert Smolka
	Prof. Dr. Andrej Bauer
	Prof. Dr. Yves Bertot
	Prof. Dr. Bas Spitters
	Dr. Andreas Buchheit (Academic Staff)

Abstract

We give a formalised and machine-checked account of computability theory in the Calculus of Inductive Constructions (CIC), the constructive type theory underlying the Coq proof assistant.

We first develop synthetic computability theory, pioneered by Richman, Bridges, and Bauer, where one treats *all* functions as computable, eliminating the need for a model of computation. We assume a novel parametric axiom for synthetic computability and give proofs of results like Rice's theorem, the Myhill isomorphism theorem, and the existence of Post's simple and hypersimple predicates relying on no other axioms such as Markov's principle or choice axioms.

As a second step, we introduce models of computation. We give a concise overview of definitions of various standard models and contribute machine-checked simulation proofs, posing a non-trivial engineering effort.

We identify a notion of synthetic undecidability relative to a fixed halting problem, allowing axiom-free machine-checked proofs of undecidability. We contribute such undecidability proofs for the historical foundational problems of computability theory which require the identification of invariants left out in the literature and now form the basis of the Coq Library of Undecidability Proofs.

We then identify the weak call-by-value λ-calculus L as sweet spot for programming in a model of computation. We introduce a certifying extraction framework and analyse an axiom stating that every function of type $\mathbb{N} \to \mathbb{N}$ is L-computable.

Zusammenfassung

Wir behandeln eine formalisierte und maschinengeprüfte Betrachtung von Berechenbarkeitstheorie im Calculus of Inductive Constructions (CIC), der konstruktiven Typtheorie die dem Beweisassistenten Coq zugrunde liegt.

Wir entwickeln erst synthetische Berechenbarkeitstheorie, vorbereitet durch die Arbeit von Richman, Bridges und Bauer, wobei *alle* Funktionen als berechenbar behandelt werden, ohne Notwendigkeit eines Berechnungsmodells. Wir nehmen ein neues, parametrisches Axiom für synthetische Berechenbarkeit an und beweisen Resultate wie das Theorem von Rice, das Isomorphismus Theorem von Myhill und die Existenz von Post's simplen und hypersimplen Prädikaten ohne Annahme von anderen Axiomen wie Markov's Prinzip oder Auswahlaxiomen.

Als zweiten Schritt führen wir Berechnungsmodelle ein. Wir geben einen kompakten Überblick über die Definition von verschiedenen Berechnungsmodellen und erklären maschinengeprüfte Simulationsbeweise zwischen diesen Modellen, welche einen hohen Konstruktionsaufwand beinhalten.

Wir identifizieren einen Begriff von synthetischer Unentscheidbarkeit relativ zu einem fixierten Halteproblem welcher axiomenfreie maschinengeprüfte Unentscheidbarkeitsbeweise erlaubt. Wir erklären solche Beweise für die historisch grundlegenden Probleme der Berechenbarkeitstheorie, die das Identifizieren von Invarianten die normalerweise in der Literatur ausgelassen werden benötigen und nun die Basis der *Coq Library of Undecidability Proofs* bilden.

Wir identifizieren dann den call-by-value λ-Kalkül L als *sweet spot* für die Programmierung in einem Berechnungsmodell. Wir führen ein zertifizierendes Extraktionsframework ein und analysieren ein Axiom welches postuliert dass jede Funktion vom Typ $\mathbb{N} \to \mathbb{N}$ L-berechenbar ist.

Acknowledgements

This thesis is a product of more than seven years of research. Since then, my journey through academia has brought me to many great places and introduced me to many great people. I feel that the final product of the journey – this thesis – is a result of a big collaboration of all the people who journeyed with me.

Ein riesiger Dank geht an meine Familie. Papa, ich hab die letzten Jahre häufig gemerkt wie viel ich von dir gelernt habe, und wie viel ich noch lernen kann. Die Leichtigkeit, die du zeigst, während du jeden Morgen so früh aufstehst, und die Energie, die du trotzdem für dein Umfeld hast, sind mir ein Vorbild. Mama, deine Liebe, dein Interesse, deine Unterstützung, dein Verständnis und dein Vertrauen waren unabdingbar für mich. Joshua, danke dass du mich in FIFA bisweilen hast gewinnen lassen. Opa, es macht mich immer noch traurig, dass du meinen Werdegang nicht begleiten konntest. Danke für das Fördern meiner Neugier und dein Vertrauen in meine Fähigkeiten ab dem ersten Tag. Ich wäre ohne euch alle nie an den Punkt gekommen, überhaupt diese Dissertation zu schreiben.

Mi familia peruana, gracias por todo. Saskia, Gabriel, Ximena, Miguel, Daniel, Priscilla, Raphita, Yuanse, Hans, Fanny, Guapo, Veronika, Walter, Nimia: Danke für eure Fröhlichkeit, eure Toleranz und eure Liebe. Danke, dass ihr mich aufgenommen habt, für die vielen vielen schönen Erinnerungen und danke dass ihr mich behalten habt. Los quiero mucho! Besonderer Dank an dieser Stelle auch an Saskia und Toni für Jolly Jumper.

Kristin, Christina, Sina, Leah, Ricardo, Peter, Leonie, Lynn, Taek, Miran, Nicola, Joel, Adéle, Susanne, Anela und Maral: Danke für die tolle Zeit in der Arndtstraße, danke für die vielen schönen Abende und das viele gute Essen, danke für euer Verständnis, wenn ich wieder wochenlang nicht da war, und danke für eure Geduld, wenn ich wochenlang da war aber trotzdem nicht das Bad geputzt hab. Ich hoffe der Punkt, dass ich hier 16 (!) Menschen danken muss ist kein Hinweis darauf, dass ich euch ein schlechter Mitbewohner. Ihr wart alle großartig! Peter, danke dass du mein Home-Office-Büropartner warst und für deine Unterstützung – nicht nur dabei pünktlich beim Frühstück zu erscheinen. Anela, du fehlst in der Arndtstraße. Immer wenn ich friere denke ich an dich und sage mir "damals haben wir jahrelang ohne Heizung gelebt, das jetzt ist halb so wild". Et pour les derniers mois, merci beaucoup à notre coloc Annick! Ganz besonders danken möchte ich auch Christopher und Wolfang für die regelmäßige Beherbergung in den ersten Jahren meines Studiums.

Den Kopf frei bekommen habe ich in Saarbrücken immer durch Fußball, Theater und den Lesekreis.

Markus, Dirk, Andreas, Lukas, Amer, Timo, Julian, Hazem, Gastón, Torben, Miran, Rayan, Mustafa, Lena, Tristan, Peter, Nico, Jan, Roland, Alex, Jasmin: Danke, dass ihr euch die Grätschen erspart habt und dass ihr alle genauso lauffaul wart wie ich (naja, fast alle). Ich werde viele Dinge an Saarbrücken vermissen, aber mit euch zu kicken vielleicht am meisten. Ein ganz besonderer Dank geht an Julian. Danke für deine Freundschaft, für die Abende im Viertel, in Rio und in Cumbuco, für die Triftparties und Klavierkonzerte, danke dass du mich den Fußballern und dem Lesekreis vorgestellt hast. Kein Dank für den gelegentlichen Tritt ans Schienbein, aber das habe ich gern in Kauf genommen. Für die Zukunft, das sag ich hier ganz öffentlich, hast du eine Sylvester-Cage-Flatrate bei mir.

An alle Thunis-Mitstreiter, die mit mir auf der Bühne standen, improvisiert haben oder in der Nautilus-Bar gefeiert haben: Danke für eure Energie, für eure Kreativität und eure Liebe zum Theater. Ganz besonders hervorheben möchte ich Renée, Peter und Julia: Danke für eure Freundschaft über die Jahre!

Ksenija, Mimi, Julian, Franzi, Tristan, Dominik, Newsha, Lena, Jana, Judith: Der Lesekreis hat mich zurück zum Lesen gebracht, und allein dafür bin ich euch dankbar. Danke dafür wie offen ihr eure Gedanken geteilt habt und wie sehr ich dadurch meinen Horizont erweitern und meinen Blick auf die Welt verbessern konnte.

Danke auch an all die anderen Freunde nah und fern. Danke, dass ihr immer verstanden habt, dass eine WhatsApp-Antwort mal länger dauern kann und danke, dass ihr mich über all die Jahre begleitet habt. Besonderer Dank dafür an meinen ältesten Freund Jan und an Lena! Lastly, a big thank you to all the people who made arriving in Paris so pleasant.

Academically, I was incredibly lucky to be part of various amazing communities.

The special atmosphere that I experienced at Saarland University especially during my first years was second to none. To the Al Bacio-Crew, the Vorkurs-Team, the various tutor teams I was part of, and to the team of Programming 1 18/19 I send huge thanks! Chris, Clara, Jana, Sebastian: Thank you for your friendship, for keeping me informed about all the small and big things in the world, and for your wisdom in all things concerning Hühnerzucht. Chris, it was invaluable for me to have somebody to talk to who was feeling the same things and having the same problems in the last weeks. Prof. Hermanns, thank you for your support in my first years and the amazing recommendation letters. Andrej, Chris, Dominik, Dominique, Fabian, Felix, Marc, Nils, Kathrin, Simon: Thanks for reading my thesis, for the invaluable fresh perspectives, and the many, many, many typos I never would have spotted.

The theorem proving, programming languages, and Coq community all were incredibly welcome and open. I have learned a great deal from many people, and I want to thank you all. I especially want to thank the Coq developers and attendees of the Coq Users and Developers Workshops for the great atmosphere. I always found that a particular pleasure of working in this field is that approaching the "big names" is possible. Dana Scott and Thierry Coquand

helped me out with historical reference and valuable advice, thank you! I also want to thank Thorsten Altenkirch, Andrej Bauer, Jasmin Blanchette, Mario Carneiro, Hugo Herbelin, Cătălin Hriţcu, Jean-François Monin, Pierre-Marie Pédrot, and Thomas Streicher for discussions and comments, some of which turned out to be essential for my thesis. Special thanks to Andrej Bauer, Yves Bertot, and Bas Spitters for reading and reviewing this long piece of work.

I had the huge luck to find many people mentoring and advising me. Ohad, thank you for your wisdom, your guidance, your hospitality, and your continued interest. Dominique, thank you for joining me in my synthetic undecidability endeavours, for your hospitality, and for your endurance regarding seemingly impossible Coq proofs. Matthieu, thank you for introducing me to the MetaCoq and CertiCoq teams, thank you for your friendship, and thank you for the occassional beer or two in great places. Nicolas, thanks for your support in the last year. I'm looking forward to spending the next years in Nantes!

DBLP tells me that I published papers and pre-prints with 25 co-authors in the last years, and I'm sure it counts better than I do. I also was lucky to work with and learn from the people in the MetaCoq and CertiCoq teams. Fabian, Gert, Dominik, Ohad, Sam, Dominique, Matija, Dominik, Maximilian, Nicolas, Simon, Simon, Kathrin, Théo, Marc, Edith, Matthieu, Bohdan, Gregory, Steven, Abhishek, Cyril, Marcel, Florian, Felix, Andrew, Zoe, Joomy, John: I learned something from each and every one of you. Thank you for your energy, for your ideas, and for the great time I had working with you.

Lastly, I was incredibly lucky to be part of Prof. Smolka's group in Saarbrücken. Gert, Andrej, Christian, Dominik, Fabian, Jonas, Kathrin, Moritz, Sigurd, Steven: Thanks for your interest in literally any topic in the world of logic and type theory, thanks for your curiosity, thanks for the long discussions, thanks for the many things you taught me, and thanks for the many memories I will keep from travelling to conferences with you. Very special thanks go to Fabian, for working so closely with me for so many years. Edith, Maximilian, Simon, Dominik, Marcel, Felix, Roberto: It was an absolute pleasure working with you. I'm very lucky to be able to say that I learned a lot from everyone of you, not only scientifically. Ute, I always highly appreciated the all-inclusive care, the chats at the water heater, and that we were spared the stress regarding all administrative things.

Gert, I struggle to express how thankful I am. Your energy and amazement in every single conversation was inspiring. For more than eight years I knew that I had an adviser and mentor that I could turn to with every matter. Thank you for teaching me that further simplification is always possible and worthwhile, for tirelessly explaining the world of research and its hidden pitfalls, for reminding me of the Wittgenstein quote "Alles, was sich aussprechen lässt, lässt sich klar aussprechen." when I needed it, and for following this motto in your criticism. I can't imagine a Doktorvater who better masters advising on the full scale from letting complete freedom to offering close collaboration.

Der größte Dank geht an meine Partner der letzten Jahre.

Saskia, du hast mich dahin gebracht wo ich heute bin und mich zu dem gemacht der ich heute bin. Ich hab viel gelernt, gesehen und verstanden durch dich. Deine Herzlichkeit, dein unbändiger Wille und dein Ehrgeiz bleiben mir ein Vorbild. Wenn ich gezweifelt hab warst du immer da um mich zu bestärken, und wenn ich mir meiner selbst zu sicher war, hast du gebremst. Du weißt selbst am Besten wie groß dein Anteil an allem war. Danke.

Dominik, dein unerreichtes Talent zu erkennen, wann eine Situation Leichtigkeit benötigt, wo ich gerade zu ernst war, und wann eine Situation ernst genommen werden muss, wo ich sie gerade unterschätzt habe, hat mir unglaublich oft geholfen. Danke, dass ich mich drauf verlassen konnte, dass da jemand zum reden ist, danke für die unvergesslichen Reisen, die vielen Abende, für das Bier, die Zauberbohnen, die Triftparties, für dein Interesse an allem drumherum und an meinem Bizeps. I think nobody apart from Gert has shaped my view on research, mathematics, and writing as much as you did. Thanks for repeating what you said when I accidentally stopped listening, thanks for your many many explanations at the whiteboard, thanks for the lunch conversations, and thanks for occasionally leaving me the sofa after lunch. Also thank you for teaching me many space-saving tricks in LaTeX. I had to use them all for these acknowledgements. Lastly, thank you for reminding me that scientific texts are allowed to sound good occasionally. The fact that this thesis contains the words "henceforth" and "moreover" is solely due to you. To keep up a tradition I let one "overview over" remain in the thesis. Happy searching!

Ksenija, danke dass du mich nimmst wie ich bin, aber auch dafür dass du immer siehst wie ich sein will und alles dafür gibst mich darin zu unterstützen. Ich hab vor kurzem den klugen Satz gelesen, dass "ein Mensch, der etwas leistet, das nie allein tut. Er kann nur sein, wer er ist, wegen der Menschen, die ihn lieben: [...] Jene Menschen, die sich jeden Abend den Wust aus Halbgarem anhören, den jemand erzählt, um seine Entscheidungen vorzusortieren."[1] Ich kann's nicht besser sagen. Ich hätte das letzte Jahr voller Lockdowns, Quarantäne und Schreibfrust so nie ohne dich, dein Zuhören, deine Hilfe beim Vorsortieren meiner Entscheidungen, deinen Rat und deine Liebe durchstehen können. Durch dich werde ich diese Zeit jetzt aber schlicht als wunderbar in Erinnerung behalten. Danke für deine Ruhe, dein Verständnis in jeder Lage, deine Geduld und für deinen Mut, mit mir gemeinsam neue Wege zu gehen.

[1] Falls ich mal Politiker werde, hier die saubere Zitation: Fritzsche, Lara: *Achterbahnfahrt für die Zunge.* In: Süddeutsche Zeitung Magazin, Heft 28/2021. Abgerufen am 21. Juli 2021. https://sz-magazin.sueddeutsche.de/getraenkemarkt/gruener-tee-noma-redzepi-90420

Contents

CHAPTER 1

Introduction

Most textbooks on computability start by introducing a model of computation and then develop an abstract structural theory of computability and concrete undecidability results. Proofs are usually presented on an informal level, with many references to the Church-Turing thesis to avoid spelling out programs in the chosen model. Research papers do not differ vastly in this regard.

However, if a reader or reviewer wants to check such proofs carefully for correctness, every invocation of the Church-Turing thesis has to be checked individually. To do so in full formality, one would have to construct a program in the chosen model of computation. As a consequence, paper proofs in computability have a second, orthogonal kind of omitted details in a proof, in addition to the usual logical arguments authors deem easy enough to reconstruct for readers to leave them out. Recovering the details regarding such constructions is largely "a routine chore" (as already acknowledged in the seminal work by Post [189]), but induces considerable overhead when working in an interactive proof assistant, where all details have to be provided.

[189] Post. 1944. Recursively enumerable sets of positive integers and their decision problems.

While many areas of mathematics have been machine-checked in proof assistants like Coq, Lean, Isabelle, HOL, Agda, etc., the main hinderance for computability theory to catch up is the tedium of giving explicit constructions in models of computation, which is added on top of the reconstruction of proof arguments and general enough invariants to allow a proof by induction usual for machine-checked proofs. For involved algorithmic arguments, we estimate that the explicit constructions add one to two orders of magnitude in proof engineering time and lines of code. The most ambitious takes on machine-checked computability are by Norrish [178], Forster and Smolka [83], Carneiro [26], and Ferreira Ramos et al. [196, 67]. They all construct universal machines and reach Rice's theorem, but do not go considerably further.

[178] Norrish. 2011. Mechanised Computability Theory.

[83] Forster and Smolka. 2017. Weak Call-by-Value Lambda Calculus as a Model of Computation in Coq.

[26] Carneiro. 2019. Formalizing Computability Theory via Partial Recursive Functions.

[196] Ramos et al.. 2018. Formalization of the Undecidability of the Halting Problem for a Functional Language.

[67] Ferreira Ramos et al.. 2020. Formalization of Rice's Theorem over a Functional Language Model.

The subfield of computability theory with the greatest influence on present-day computer science is likely formed by undecidability proofs for concrete problems with applications like type-checking, planning, unification, satisfiability, or provability. Both decidability and undecidability proofs rely on intricate details which are hard to check manually and various subtle errors in such proofs have been discovered.[1] Furthermore, natural questions in the area remain open, like whether solvability of polynomials over rational numbers is decidable. While machine-checked decidability proofs can be found in the literature [58, 163, 203, 57], we are not aware of machine-checked *un*decidability results for natural problems independent of

[1] E.g. in 1932 Gödel claimed without proof that his decidability proof for the $[\exists^*\forall^2\exists^*, \mathrm{all}, (0)]$ fragment of first-order logic could be extended to include equality [101], but the extended fragment was proved undecidable in 1984 by Goldfarb [98, 97]. Kfoury, Tiuryn, and Urzyczyn [126] remark in their paper on the undecidability of semi-unification in 1993 that "among the many erroneous claims announcing the decidability of [semi-unification] there was also ours [125]" from 1988. A result by Ghelli [90] from 1990 implying the decidability of type-checking and subtyping in $F_{\leq}$ [43] was found to be erroneous by Ghelli himself [91] and superseded by an undecidability proof by Pierce [185] in 1994. The decidability proof for the MELL-fragment of linear logic [22] from 2015 was disputed by Straßburger [219] in 2019, leaving the status of the problem unresolved.

models of computation,lap.[2] i.e. problems other than halting problems or problems subject to Rice's theorem.

This thesis takes a radically different approach compared to textbooks, ultimately leading to feasible approaches for a machine-checked theory of computability and machine-checked undecidability proofs, both without the overhead of verifying programs in models of computation. All proofs in the thesis are formalised in a computational, constructive type theory: The Calculus of Inductive Constructions (CIC) [182] underlying the Coq proof assistant [222]. Consequently, all proofs are machine-checked by Coq.

[182] Paulin-Mohring. 2015. Introduction to the Calculus of Inductive Constructions.

[222] The Coq Development Team. 2021a. The Coq Proof Assistant version 8.13.2.

A machine-checked theory of computability constitutes a further step in the overarching goal of mechanising the pillar-stones of mathematics and computer science. The mechanisation in a proof assistant enforces the isolation of clear invariants, which are often left out in the literature. Thus, we contribute to a better understanding and clearer presentation of these often-taught topics. Additionally, with the help of Coq we obtain a simplification of proofs and logical assumptions: our proofs cover every detail, but are often still easier to convey than textbook arguments, while additionally being fully constructive.

Machine-checked undecidability proofs complement machine-checked decidability proofs and could be used to settle debates regarding future and past publications of (un)decidability results since the machine-checked evidence is undebatable.

By basing the development of computability theory in constructive type theory, a foundational system agnostic towards axioms like the law of excluded middle, the textbook strategy can be reversed: We first develop computability theory based simply on *functions* rather than *computable* functions – a so-called "synthetic approach". Since Kleene's work on realizability in the 1940s, it is well-known that such an approach is possible in constructive logic, since all constructively definable functions of type $\mathbb{N} \to \mathbb{N}$ *correspond* to programs in some model of computation. The synthetic approach to computability is arguably most natural in constructive type theory, where every function *is* a program in a variant of the λ-calculus. Only afterwards, we introduce models of computation, show them Turing-equivalent, and present undecidability proofs for non-computational problems from logic and theoretical computer science.

In Part I, we introduce the basic notions of synthetic computability like decidability, enumerability, semi-decidability, and reducibility. As is natural in type theory, the notions are based on predicates rather than sets. We prove basic results like Post's theorem ("enumerable predicates with enumerable complement are decidable") or the Myhill isomorphism theorem ("one-one equivalent predicates are recursively isomorphic"), where the synthetic approach allows us to focus only on the mathematical essence of the theorems.

Since CIC is consistent with axioms entailing the synthetic decidability of any problem, proofs showing the absence of a decision function for problems are impossible in CIC without assuming axioms. We thus assume a parametric form of synthetic Church's thesis (CT) we call SCT, postulating a step-indexed interpreter parametrically universal for functions $\mathbb{N} \to \mathbb{N}$. The parametric axiom is strong enough to derive all results and in particular an S_n^m-like theorem. The strict separation between propositions and computation in CIC which renders virtually no choice principle provable furthermore makes the resulting synthetic theory of computability agnostic towards axioms like the law of excluded middle or the axiom of choice: It does not rely on them and moreover both are respectively consistent with the theory. This stands in contrast to the pioneering work in synthetic computability by Richman, Bridges, and

[2] and other than those building on the results developed as part of this thesis.

Bauer [198, 25, 10], where choice principles are assumed and the law of excluded middle can be disproved.

[198] Richman. 1983. Church's thesis without tears.

[25] Bridges and Richman. 1987. Varieties of constructive mathematics.

[10] Bauer. 2006a. First steps in synthetic computability theory.

To demonstrate the versatility and elegance of the resulting theory based on CIC and the parametric axiom, we give proofs of the undecidability of the synthetic halting problem, Rice's theorem, or the existence of (hyper)simple predicates solving Post's problem for many-one and truth-table reducibility.

In Part II, the thesis considers various models of computation. We cover λ-calculus, Turing machines, binary stack machines, counter machines, FRACTAN, and μ-recursive functions.

We provide simulation proofs between the models, establishing them all to be Turing-equivalent. We also consider Diophantine equations and show a form of the DPRM theorem, i.e. show that a relation is Diophantine if and only if it is computable in any of the models of computation considered. The formal techniques used span a variety of approaches like interpretation via an elaborated abstract machine, compilation and linking, or the arithmetisation of computation. All simulation proofs are fully constructive. As a result, we obtain synthetic many-one reductions between the respective halting problems.

In Part III, we identify a notion of synthetic undecidability relative to a halting problem for one of the models from Part II. Synthetic undecidability can be established without the assumption of any axioms while still avoiding the verification of programs in models of computation. Thus, synthetic undecidability proofs can focus on the essential reasons a problem is undecidable and allow the isolation of inductive invariants.

We give machine-checked synthetic undecidability results for concrete problems by verifying many-one reduction chains starting at halting problems: (Semi-Thue) string rewriting, the word problem for semi-groups, the Post correspondence problem, provability, validity, and satisfiability for intuitionistic and classical first-order logic, the intersection and palindrome problems for context-free grammars, and higher-order unification in the simply-typed λ-calculus are shown undecidable. These problems constitute the basis for the Coq Library of Undecidability Proofs, a collaborative effort with many contributors, presenting existing and novel undecidability proofs in a unified library of more than 100.000 lines of code.

In Part IV, the weak call-by-value λ-calculus L is singled out as sweet-spot for the formalisation and mechanisation of concrete results. By developing a certifying extraction to L from a simply-typed subset of Coq, L becomes a powerful tool to close loops for many-one equivalence and Turing-equivalence proofs. We conclude by discussing the axiom CT_{L}, stating that any function of type $\mathbb{N} \to \mathbb{N}$ is L-computable, in connection to all parts of the thesis.

The four parts of the thesis have individual, detailed introductions and outlines. Parts I to III can be read in any order. In particular, parts II and III are written in an encyclopedic style, presenting a unified mathematical introduction to the problems and proofs in the Coq Library of Undecidability Proofs [82].

[82] Forster, Larchey-Wendling, Dudenhefner, Heiter, Kirst, Kunze, Smolka, Spies, Wehr, and Wuttke. 2020d. A Coq library of undecidable problems.

1.1 Publications

Parts of the thesis are based on the following publications:

[71] Forster, Heiter, and Smolka. "Verification of PCP-Related Computational Reductions in Coq." *ITP 2018*.

[147] Kunze, Smolka, and Forster. "Formal small-step verification of a call-by-value λ-calculus machine." *APLAS 2018*.

[84] Forster and Smolka. "Call-by-value lambda calculus as a model of computation in Coq." *Journal of Automated Reasoning*. 2019.

[81] Forster and Larchey-Wendling. "Certified Undecidability of Intuitionistic Linear Logic via Binary Stack Machines and Minsky Machines." *CPP 2019*.
[73] Forster, Kirst, and Smolka. "On synthetic undecidability in Coq, with an application to the Entscheidungsproblem." *CPP 2019*.
[152] Larchey-Wendling and Forster. "Hilbert's Tenth Problem in Coq." *FSCD 2019*.
[76] Forster and Kunze. "A certifying extraction with time bounds from Coq to call-by-value λ-calculus." *ITP 2019*.
[211] Sozeau, Anand, Boulier, Cohen, Forster, Kunze, Malecha, Tabareau, and Winterhalter. "The MetaCoq Project." *Journal of Automated Reasoning*. 2020.
[77] Forster, Kunze, and Roth. "The weak call-by-value λ-calculus is reasonable for both time and space." *POPL 2020*.
[79] Forster, Kunze, and Wuttke. "Verified programming of Turing machines in Coq." *CPP 2020*.
[216] Spies and Forster. "Undecidability of higher-order unification formalised in Coq." *CPP 2020*.
[74] Forster, Kirst, and Wehr. "Completeness Theorems for First-Order Logic Analysed in Constructive Type Theory." *LFCS 2020*.
[153] Larchey-Wendling and Forster. "Hilbert's Tenth Problem in Coq (extended version)." *Logical Methods in Computer Science*. 2020.
[70] Forster. "Church's thesis and related axioms in Coq's type theory." *CSL 2021*.
[78] Forster, Kunze, Smolka, and Wuttke. "A Mechanised Proof of the Time Invariance Thesis for the Weak Call-by-value λ-Calculus." *ITP 2021*.

Furthermore, some results can be found in the following pre-print:

[72] Forster, Jahn, and Smolka. "A Constructive and Synthetic Theory of Reducibility: Myhill's Isomorphism Theorem and Post's Problem for Many-one and Truth-table Reducibility in Coq." 2021.

The author of this thesis was the main author of [84, 79, 70, 72] and advised the Bachelor theses of Edith Heiter [103], Maximilian Wuttke [238], Simon Spies [215], Dominik Wehr [233], and Felix Jahn [124]. More information on the connections of the publications on the thesis can be found in the introduction of every respective chapter.

1.2 Contributions

Many contributions in the thesis were conceived, formalised, and mechanised in joint work, indicated by the cited paper being co-authored.

Part I We present the first formalised and machine-checked study of synthetic computability in type theory.

- We introduce decidability, enumerability, and semi-decidability and analyse their relationship on arbitrary base types [73].
- We introduce one-one, many-one, and truth-table reducibility and prove standard results from textbooks. Results like the characterisation of truth-table reducibility as one-one reducibility are simplified in comparison to textbook proofs [72].
- We give a fully synthetic proof of the Myhill isomorphism theorem, without any axioms, and extend it to a computational Cantor-Bernstein theorem stating that discrete enumerable types X, Y with injections $X \to Y$ and $Y \to X$ between them admit an isomophism, i.e. two functions $X \to Y$ and $Y \to X$ inverting each other [72].
- We present the axioms SCT and EA as parametric synthetic versions of CT, which enable formalised and machine-checked synthetic computability theory compatible with classical intuition [70].
- We survey the computational status of axioms in CIC, especially in relation to CT. We observe that the strict separation of proposition and computation in CIC makes virtually no

choice principles provable and thus allows the consistent assumption of the law of excluded middle together with CT [70].
- We develop synthetic computability theory, to an extent of roughly an introductory lecture to computability.
- We give two synthetic proofs of Rice's theorem, based on different formulations of SCT.
- We give a synthetic solution to Post's problem for many-one reducibility, constructing a simple predicate. We give a direct proof that simple predicates are undecidable and show that an undecidability proof by many-one reduction from the halting problem is impossible [72].
- We give a synthetic solution to Post's problem for truth-table reducibility, constructing a hypersimple predicate. We give a direct proof that hypersimple predicates are undecidable and show that an undecidability proof by truth-table reduction from the halting problem is impossible [72].
- We introduce a notion of synthetic Turing reducibility and analyse that its properties crucially rely on Markov's principle.
- We show that the constructed hypersimple predicate is Turing-reducibility complete.

Part II We introduce several standard models of computation in a unified formal setting: the weak call-by-value λ-calculus L, multi-tape and single-tape Turing machines, binary stack machines, counter machines, FRACTRAN, Diophantine equations, and μ-recursive functions. We prove the Turing completeness of all of these models, by showing that their halting problems are inter-reducible and that relations $R\colon \mathbb{N}^k \to \mathbb{N} \to \mathbb{P}$ are computable in one of the models if and only if they are in all other. Technically, the contributions lie in finding the right abstractions and invariants to enable proofs by induction, which are usually both left out in textbooks. More precisely, we:

- mechanise a definition of Turing machines in Coq [79], based on a formalisation by Asperti and Ricciotti in Matita.
- survey a verification framework for Turing machines [79]. The framework also has tools for the verification of time and space complexity, which we do not cover.
- formalise and mechanise the first verified translation of a λ-calculus to Turing machines. Specifically, we formalise and mechanise a simulation of the weak call-by-value λ-calculus L on Turing machines using a stack machine semantics for L. The stack machine semantics was developed mainly by Fabian Kunze and Gert Smolka [147]. The Coq verification of the resulting Turing machines was carried out mainly by Maximilian Wuttke and Fabian Kunze, with contributions by the author of the thesis [78].
- mechanise a translation of multi-tape Turing machines to single-tape Turing machines [79], based on a formalisation in the book of Sipser. The Coq verification of the Turing machines was carried out mainly by Maximilian Wuttke.
- formalise and mechanise a translation of Turing machines with arbitrary alphabet to one with binary alphabet.
- formalise and mechanise a translation of single-tape binary Turing machines to binary stack machines.
- extend the translation of binary stack machines to counter machines due to Dominique Larchey-Wendling [81] to a full simulation result.
- formalise and mechanise a translation from counter machines to FRACTRAN [152].
- survey the synthetic undecidability proof of Hilbert's tenth problem, carried out by Dominique Larchey-Wendling with contributions by the author of this thesis [152].

- formalise and mechanise a reduction of solvability of Diophantine equations to solvability of polynomial integer equations via Lagrange's theorem[153].
- extend the reduction from solvability of Diophantine equations to the halting problem of μ-recursive functions due to Dominique Larchey-Wendling to a full simulation result [153].

Part III We introduce the notion of synthetic undecidability, which enables undecidability proofs in Coq not relying on axioms or models of computation [81, 73]. We give machine-checked synthetic undecidability proofs for most of the prominent historical foundational problems in the literature. Technically, the contributions lie in finding the right abstractions and invariants to enable proofs by induction, which are usually left out in textbooks. We cover:
- Post's correspondence problem [71],
- (semi Thue) string rewriting [71],
- the word problem for semi-groups (also known as Thue rewriting),
- (naive) Tarski-style validity and minimal, intuitionistic, and classical provability for first-order logic [73],
- the intersection problem and palindrome problem of context-free grammars [71],
- higher-order unification in the Curry-style simply-typed λ-calculus [216].

We report on the creation of the Coq Library of Undecidability Proofs, together with Dominique Larchey-Wendling.

Part IV We identify the weak call-by-value λ-calculus as sweet spot for developing concrete results in a model of computation [77]. We introduce an extraction framework yielding certified terms in L for simply-typed, non-dependent Coq functions [76, 211]. The tactics for the automatic verification of terms and the semi-automatic time complexity verification are not covered and are due to Fabian Kunze [76]. We use the certifying extraction framework for the following results:
- We verify a simulation of μ-recursive functions in L [153].
- We verify a simulation of Turing-machines in L [78]. We do not cover the time complexity of the simulation, which is due to Fabian Kunze.
- We verify a reduction of first-order provability to the halting problem of L [74].
- We verify a reduction of the Post correspondence problem to L.

Lastly, we verify the S_n^m theorem for L and discuss several equivalent formulations of CT using L and other models of computation.

1.3 Mechanisation in Coq

The mechanised proofs of Part I are in a repository on GitHub:

`https://github.com/uds-psl/coq-synthetic-computability`

The results of Part II, III, and IV are contributed or in the process of being contributed to the Coq Library of Undecidability Proofs. All results can also be found here:

`https://ps.uni-saarland.de/~forster/thesis`

Comments on length and authorship of the respective code can be found in the respective introductions of the four parts of the thesis.

All code compiles with The Coq Proof Assistant version `8.13.2`.

The central statements in the pdf of this thesis are hyperlinked with the html-version of the Coq code, indicated by a clickable -symbol. Furthermore, symbols in sf font are hyperlinked with their definitions.

Part I
Synthetic computability

CHAPTER 2

Introduction: Synthetic Computability

Constructive type theory as a foundational system is built on top of a λ-calculus with inductive types, making computation native. Thus, it is natural to treat the function space $\mathbb{N} \to \mathbb{N}$ as the type of *computable* functions, and consequentially to define computational notions such as decidability and enumerability in terms of such functions, without mentioning any traditional model of computation.

Such definitions can be seen as *synthetic*. In synthetic approaches to an area of mathematics, the objects of the logic (which, in our case, is constructive type theory) are turned into the structures under investigation, often by assuming suitable axioms on the objects of the logic. Synthetic approaches thereby use the axiomatic freedom provided by constructive foundations to the fullest. In synthetic computability, one assumes axioms on functions – the objects of the logic – to ensure they behave like computable functions – the structures under investigation.

In contrast, textbook computability theory is *analytic*. In analytic approaches, the objects of the logic are used to model the structures under investigation, often via multiple layers of encodings and abstraction. Thus, in analytic computability theory, one defines models of computation, defines which parts of the model are considered programs, defines the behaviour of these programs, defines encodings for e.g. natural numbers as data, defines when a function is computed by a program, etc.

This first part of the thesis is devoted to the development of basic results in computability theory in a synthetic way. It covers the content of about an introductory lecture to computability. The synthetic approach to computability enables a clear view on the essence of proofs, since no distracting routine manipulations of encodings or models are necessary. As such, it is well-suited for mechanisation: We formalise all our proofs in the Calculus of Inductive Constructions (CIC), the type theory underlying the Coq proof assistant, and all proofs are machine-checked by the Coq proof assistant.

Synthetic computability originated in the works of Richman and Bridges [198, 25] for recursive analysis and was developed more generally by Bauer [10, 13, 14]. Richman, Bridges, and Bauer work in Bishop-style constructive logic with at least the axiom of countable choice and assume suitable axioms for synthetic computability. Countable choice is used as replacement for the S_n^m theorem in many results, and thus integral to the development. As a result, the assumed axioms on synthetic computability render the theory anti-classical, meaning the law of excluded middle $\forall P.\ P \vee \neg P$ is provably wrong. Classical proofs are however omnipresent in analytic treatments of computability, where the law of excluded middle is routinely used. Thus, synthetic computability à la Richman, Bridges, and Bauer is incompatible with classical textbook intuitions. This might have contributed to synthetic computability being a niche of constructive mathematics rather than the most natural way to carry out computability theory.

We first cover some standard results without appealing to axioms at all. We introduce decidability, enumerability, and semi-decidability as well as one-one, many-one and truth-table reducibility, and prove standard results like the Myhill isomorphism theorem or the characterisation of many-one and truth-table reducibility in terms of one-one reducibility.

[198] Richman. 1983. Church's thesis without tears.

[25] Bridges and Richman. 1987. Varieties of constructive mathematics.

[10] Bauer. 2006a. First steps in synthetic computability theory.

[13] Bauer. 2017. On fixed-point theorems in synthetic computability.

[14] Bauer. 2020. Synthetic mathematics with an excursion into computability theory (slide set).

To show undecidability, non-enumerability, or non-reducibility results, we assume axioms for synthetic computability. Precisely, we assume SCT, a parametric version of Richman's synthetic formulation of the constructivist axiom CT[1], and an analogous parametical version of Bauer's enumerability axiom, which we call EA. While CT states that every function $\mathbb{N}\to\mathbb{N}$ is μ-recursive, SCT postulates a step-indexed interpreter ϕ which is *parametrically* universal for the function space $\mathbb{N}\to\mathbb{N}$. This means that for every family of functions $f_i\colon\mathbb{N}\to\mathbb{N}$ for $i\colon\mathbb{N}$, one obtains a coding function $\gamma\colon\mathbb{N}\to\mathbb{N}$ such that $\phi_{\gamma i}$ and f_i agree. SCT can equivalently be formulated as the axiom EA, postulating an enumerator φ which is parametrically universal for all enumerable predicates $\mathbb{N}\to\mathbb{P}$. That means that for every family of predicates $p_i\colon\mathbb{N}\to\mathbb{N}$ for $i\colon\mathbb{N}$ which has a parameterised enumeration function f such that f_i enumerates p_i, one obtains a coding function $\gamma\colon\mathbb{N}\to\mathbb{N}$ such that $\varphi_{\gamma i}$ enumerates p_i. While SCT is conceptually closer to the well-studied axiom CT in constructive mathematics, EA allows more elegant proofs. Both have in common that their formulation – in contrast to CT – does not require a formalised model of computation.

We build our work on the hypothesis that in CIC, the axiom CT can be consistently assumed in conjunction with the law of excluded middle LEM. This stands in contrast to set-theoretic foundations, where LEM and CT are not compatible due to the provability of unique choice. In Chapter 7 we argue the crediblility of this hypothesis, by an extensive survey of literature regarding consistency and incompatibility results. Outside of Chapter 7 we assume that the hypothesis holds without reiterating its status.

We first develop impossibility results solely based on EA: Our proof of Rice's theorem consists only of the mathematical essence, with no overhead whatsoever. We then present constructive proofs of Post's problem for many-one and truth-table reducibility, i.e. we show that there are enumerable predicates S and H such that S is undecidable but does not many-one reduce from the halting problem, since S is a simple predicate, and H is undecidable but does not truth-table reduce from the halting problem, since H is hypersimple. We then assume Markov's principle MP and show that H is Turing-reducibility complete, i.e. every enumerable predicate Turing-reduces to H.

Overall, we obtain the first machine-checked proof of the Myhill isomorphism theorem, the first machine-checked proofs of Post's problem w.r.t. many-one and truth-table reducibility, the first mechanised definition of Turing reducibility, and the most concise machine-checked proof of Rice's theorem.

Since EA assumes a *parametrically* universal enumerator, we do not have to rely on countable choice or unique choice anymore. In CIC, functions and functional relations are separate objects, and thus both forms of choice are independent in CIC. By basing it in CIC, our development of synthetic computability then becomes compatible with classical reasoning, i.e. the law of excluded middle can be consistently assumed in addition to our axioms. Thus, we claim that our approach can be considered the most natural way to formalise computability theory consistent with classical intuitions.

To support this claim we first give a brief overview of the history of both analytic and synthetic computability, before discussing the principles underlying CIC rendering it the ideal system for synthetic computability.

[1] Short for *Church's thesis*. To avoid confusion with informal Church's thesis talking about intuitive calculability we always refer to the formal axiom as CT.

History of analytic computability

The development of computability theory started in the 1930's with various proposed formalisations of computation by Post, Gödel, Turing, Church, and others. The idea that all conceivable computations can be captured by such definitions was anticipated by Emil Post in the 1920s [192], and is nowadays attributed independently to Alonzo Church, who claimed that all intuitively calculable functions $f : \mathbb{N} \to \mathbb{N}$ are μ-recursive (*Church's thesis*) [31] and Alan Turing, who claimed that all intuitively calculable functions $f : \mathbb{N} \to \mathbb{N}$ are computable by a Turing machine (*Turing's thesis*) [229]. The real matter of investigation of Church's paper is however the λ-calculus, which was proved equivalent to μ-recursive functions already before by Church, Kleene, and Rosser. Kleene [134] combined this with Turing's proof in the appendix of his paper that the λ-calculus and Turing machines are equivalent, resulting in an equivalence proof of Church's thesis and Turing's thesis, which are nowadays mostly referred to together as the *Church-Turing thesis*.

[31] Church. 1936. An unsolvable problem of elementary number theory.

[229] Turing. 1936. On computable numbers, with an application to the Entscheidungsproblem.

[134] Kleene. 1952. Introduction to metamathematics.

The Church-Turing thesis is crucially based on the notion of *intuitively calculable* functions,[2] which is inherently non-precise, but from the beginning was connected to constructive logic. Church states "this merely means that we should take the existential quantifier which appears in our definition of a set of recursion equations in a constructive sense" when discussing that every μ-recursive function is intuitively computable (i.e. Church suggests that classical proofs of μ-recursiveness might not yield intuitively computable functions). Kreisel even doubts whether "the distinction between [constructively definable] and [intuitively calculable] was recognised when Church formulated his thesis" [143, p. 143 footnote 2.].

[143] Kreisel. 1970. Church's thesis: a kind of reducibility axiom for constructive mathematics.

For constructive systems, Kleene in 1943 conjectures that all functions definable in a constructive system can be proved μ-recursive in this system (*Church's rule*) [132]. In 1945, he proves the admissibility of Church's rule for Heyting arithmetic [133], and in his 1952 book extends it to partial functions [134].

[132] Kleene. 1943. Recursive predicates and quantifiers.

[133] Kleene. 1945. On the interpretation of intuitionistic number theory.

The insights regarding the nature of computability are rightly considered ground-breaking, and necessary for computer science to arise as a subject of its own. However, if one would have to pick a founding moment of computability theory really deserving the suffix "theory", this would likely be Post's 1944 paper *Recursively Enumerable Sets of Positive Integers and Their Decision Problems* [189]. Post introduces the concept of recursively enumerable sets and analyses their relation to each other via so-called reducibilities. From the early work of Post, a rich theory of reducibility degrees, i.e. equivalence classes of inter-reducible sets, arose, and nowadays computability theorists routinely work with a multitude of reducibility notions, many of which were introduced or inspired by Post.

[189] Post. 1944. Recursively enumerable sets of positive integers and their decision problems.

Early in his paper, Post remarks "that mathematicians generally are oblivious to the importance of this work of Gödel, Church, Turing, Kleene, Rosser and others as it affects the subject of their own interest is in part due to the forbidding, diverse and alien formalisms in which this work is embodied." The evolution of "computability" to "computability theory" started by Post was enabled by a presentation of the lead questions in a more appealing, intuitive way, abstracting away from the "forbidding, alien formalisms" constituted by μ-recursive functions, Turing machines, the λ-calculus, or Post's own tag systems.

The standard textbook reference for the theory developed by and after Post is Hartley Rogers' *The Theory of Recursive Functions and Effective Computability* [202], and Rogers uses a similar approach: After introducing a universal function ϕ for μ-recursive functions, no

[202] Rogers. 1987. Theory of Recursive Functions and Effective Computability.

[2] often also called *effectively calculable* or *intuitively recursive* functions.

explicit constructions of a program in the form of a μ-recursive function is given. Instead, Rogers as well as other authors of more modern books such as Soare [210], Cutland [44], or Odifreddi [180], use informal descriptions of algorithms in prose, and then invoke the Church-Turing thesis to obtain a code c such that now ϕ_c is the representation of the algorithm as computable function. Such a quasi-axiomatic treatment of computability only based on a universal function ϕ and the Church-Turing thesis allows the authors to focus on the mathematical essence of proofs in explanations. An abstract treatment is even more crucial to prove new results, since dealing with encodings in models of computation would impede the explorative process to an extent that it becomes impossible. This is to an amount that machine-checked computability theory, where all encodings have to be given explicitly, has not been developed substantially beyond Rice's theorem.

[210] Soare. 1999. Recursively enumerable sets and degrees: A study of computable functions and computably generated sets.

[44] Cutland. 1980. Computability.

[180] Odifreddi. 1992. Classical recursion theory: The theory of functions and sets of natural numbers.

The problem can be mitigated to some extent by basing all informal investigations in constructive rather than classical logic, since all definable functions could be proved computable by applying Church's rule. However, textbooks implicitly use classical set theory with the axiom of choice, where some textbooks even announce these foundations explicitly: Rogers states that "throughout this book we allow nonconstructive methods; we use the rules and conventions of classical two-valued logic (as is the common practice in other parts of mathematics), and we say that an object exists if its existence can be demonstrated within standard set theory. We include the axiom of choice as a principle of our set theory." [202, p. 10 footnote †].

As a subfield of computability theory, *recursive analysis* (often also *computable analysis*) has got much and more recent attention in research than classical computability theory. Odifreddi [180] categorises recursive analysis into the *Markov school* and the *realm of classical logic*. In the realm of classical logic, recursive analysis makes reference to a model of computation, like standard textbooks on computability. The Markov school instead does not even consider non-computable functions. Every function is inherently an algorithm, and developing analysis becomes synonymous to developing recursive analysis.

[180] Odifreddi. 1992. Classical recursion theory: The theory of functions and sets of natural numbers.

Even more explicitly, such formal developments of recursive analysis or computability theory in constructive logic can be based on axioms: If one explicitly assumes that every function $f : \mathbb{N} \to \mathbb{N}$ is μ-recursively computable in the logic, i.e. that every application of the Church-Turing thesis to a (constructively defined) function is valid, much of computability theory can be developed without actually considering details of the model of computation. This axiom was introduced by Kreisel [142] as CT in constructive logic.

[142] Kreisel. 1965. Mathematical logic.

Synthetic computability

The axiom CT enables a synthetic development of computability theory. Philosophically, CT is however still unpleasing: To state CT and enable synthetic reasoning, an analytic formalisation of a model of computation is necessary.

The founding stone of true synthetic computability is thus Richman's 1983 paper *Church's thesis without tears* [198]: Richman identifies a fully synthetic variant of CT he calls CPF, which does not make reference to a model of computation, by only abstractly assuming the existence of a function ϕ enumerating all partial functions of type $\mathbb{N} \rightharpoonup \mathbb{N}$. Such a universal function ϕ is similar to the one used by e.g. Rogers, and indeed the developments in Richman and Rogers read virtually the same: An algorithm is given in the form of a function f, defined formally as function in the meta-theory or informally via prose. In Richman's cases, CPF immediately yields a code c such that ϕ_c extensionally agrees with f. In Rogers case, one

needs to check that f is indeed intuitively computable and apply the Church-Turing thesis to f, resulting in a code c such that ϕ_c extensionally agrees with f.

The 1983 paper of Richman is mostly concerned with presenting theorems in recursive analysis based on the axioms, e.g. he constructs Specker's sequence which yields a counterexample to the Heine-Borel theorem and to the theorem that every continuous function on the closed unit interval is uniformly continuous. In his 1987 book with Bridges [25], the authors develop more results in recursive analysis. Again the development assumes countable choice, but the authors remark that the assumption of an S_n^m operator would be sufficient for their results as well, but do not rely on this weaker assumption.

[25] Bridges and Richman. 1987. Varieties of constructive mathematics.

It is only the 2005 paper by Bauer [10] which paves the road for first steps in textbook computability based on a fully synthetic axiom. Bauer works in a topos, which can be thought of as constructive logic with extensionality axioms, Markov's principle, and the axiom of dependent and countable choice. More concretely, the topos is the effective topos [119], where the set of enumerable sets of natural numbers is enumerable. The focus on enumerable sets rather than partial functions makes Bauer's development notably more elegant.

[10] Bauer. 2006a. First steps in synthetic computability theory.

[119] Hyland. 1982. The Effective Topos.

However, due to the assumption of forms of the axiom of choice, the theory stays anti-classical: The axiom of excluded middle is provably false, and many of Bauer's theorems, while conveying interesting constructive results, become classical trivialities.

This situation can be rectified by suitably strengthening Bauer's enumerability axiom to a parametic enumerability axiom, such that choice is rendered unneccessary, and then basing the whole development on CIC, where the axiom of choice is not available. The parametric enumerability axiom is equivalent to a non-parametric invariant and the assumption of an S_n^m operator, giving more evidence for Bridges' and Richman's conjecture that this indeed suffices. Before going into detail of our setup, we first discuss the relevant aspects of CIC.

Axiomatic freedom and constructive reverse mathematics in CIC

As in most type theories, logic in CIC is modelled via the *propositions as types* paradigm based on the Curry-Howard correspondence. Specifically, in CIC propositions are types in a separate, impredicative universe of propositions $\mathbb{P}$, i.e. propositions are types $P : \mathbb{P}$, and proofs are elements $H : P$.

The logic is agnostic towards classical logical axioms like the law of excluded middle LEM. They can be consistently assumed, but are not provable, i.e. are independent. This also holds for Markov's principle MP, a consequence of LEM which is assumed in some other constructive foundations. In CIC, if one wants to use case analysis on predicates or propositions, one first has to establish that such case analyses are possible via an algorithm, i.e. prove the predicate decidable. There are three notable exceptions to this rule: If one assumes LEM, then such a case analysis is always possible. If one assumes MP, then such a case analysis is possible provided the proof goal is a semi-decidable proposition. A case analysis is always possible when one wants to prove a contradiction, i.e. when the proof goal is falsity.

Extensionality axioms like proof irrelevance ($\forall P : \mathbb{P}. \forall H_1 H_2 : P.\ H_1 = H_2$) and propositional extensionality ($\forall P_1 P_2 : \mathbb{P}.\ P_1 \leftrightarrow P_2 \to P_1 = P_2$) are similarly independent.

Lastly, almost all formulations of the axiom of choice are also independent in CIC: Total relations between types X and Y, i.e. $R : X \to Y \to \mathbb{P}$ s.t. $\forall x. \exists y.\ Rxy$, can neither be turned into functions $X \to Y$, nor into total functional relations. Not even the axiom of unique choice turning a total functional relation into a function $X \to Y$ is provable, and neither are forms of

the axiom of countable choice, where one restricts the domain X of relations to $\mathbb{N}$. This is due to the fact that propositional existence $\exists$ and computational dependent pairs Σ are strictly different objects: CIC's separation of the universe $\mathbb{P}$ from the computational universes $\mathbb{T}$ via the large elimination restriction forbids turning $\exists$ into Σ. The large elimination restriction can be seen as a barrier between propositions and computation, which can only be broken for three interesting use cases: First a proof of falsity can be used computationally, in that a program does not have to return a value for logically impossible cases. We call this a *computational explosion*. Secondly, equality proofs can be used in computations, meaning the type of a computation can be changed via a propositional equality proof, a so-called *type cast*. Thirdly, existential quantification over decidable predicates on natural numbers (i.e. $\exists n\colon\mathbb{N}.\, f\, n = \mathsf{true}$) can be computationally eliminated, yielding a function $\mu_{\mathbb{N}}\colon(\forall f.(\exists n.\, f\, n = \mathsf{true}) \to \mathbb{N})$ returning the least n satisfying $f\, n = \mathsf{true}$. We call $\mu_{\mathbb{N}}$ a *guarded minimisation* function. Guarded minimisation can be seen as a choice principle for decidable relations on countable co-domain (sometimes called Δ^0_0 choice) and also implies a choice principle for enumerable and semi-decidable relations (Σ^0_1 choice)

The independence of virtually all version of choice (despite the Δ^0_0 and Σ^0_1 versions) make CIC special in the realm of formalisations of Bishop's constructive logic. In Bishop's (non-formalised) system, a (non-extensional) form of the axiom of choice is provable, and this also holds for Martin-Löf type theory (where $\exists := \Sigma$). In less formal developments of constructive mathematics (see e.g. Diener [55] for an overview), at least countable choice if not dependent choice is always assumed. Richman critices the universal assumption of such axioms in constructive mathematics, stating that "countable choice is a blind spot for constructive mathematicians in much the same way as excluded middle is for classical mathematicians" [199, 200]. Put differently, CIC fosters an even larger axiomatic freedom than other constructive theories.

[55] Diener. 2020. Constructive Reverse Mathematics.

[199] Richman. 2000. The fundamental theorem of algebra: a constructive development without choice.

[200] Richman. 2001. Constructive Mathematics without Choice.

Axioms like the law of excluded middle or the axiom of choice and weakenings of both are central in the field of *constructive reverse mathematics* [56, 121], where one is concerned with finding sufficient and necessary axioms for results in constructive mathematics (and mostly in constructive analysis). Basing constructive reverse mathematics in a foundational system which proves the axiom of (countable) choice means that results cannot be analysed in terms of whether they can already be proved with weaker forms of choice. Thus, CIC seems to be a suitable base system for constructive (reverse) mathematics sensitive to applications of countable choice.

[56] Diener and Ishihara. 2021. Bishop-Style Constructive Reverse Mathematics.

[121] Ishihara. 2006a. Reverse Mathematics in Bishop's Constructive Mathematics.

Constructive reverse mathematics has its roots in computability theory. Most well-known, Markov's principle (formulated for μ-recursive functions) is equivalent to Post's theorem [189] that if a set and its complement are enumerable, it is decidable [228]. For other results in computability, the constructive status is unknown.

[228] Troelstra and van Dalen. 1988. Constructivism in mathematics. Vol. I.

That CT can be assumed consistently in CIC can be proved in various ways, although no published proof is available. Intuitively, this is because all three exceptions of the large elimination restriction (computational explosion, type casts, and guarded minimisation) can be given a direct computational interpretation (an arbitrary element for computational explosion, the identity function for type casts, and unbounded minimisation for guarded minimisation). As it seems, CIC's axiomatic freedom is not impacted much when assuming CT: Since propositions are strictly separated from computation, assuming the law of excluded middle does not entail non-computable functions as long as no strong forms of choice are available. Vice versa, various forms of the axiom of choice can be assumed, and the identification of functional relations with functions is safe as long as no classical totality proofs are available.

Synthetic computability retaining axiomatic freedom

To develop a synthetic approach to computability, we have to render both analytic textbook definitions and theorems in our synthetic theory. We informally write $[\![P]\!]_{\mathsf{s}}$ for the synthetic rendering of an analytic logical formula P. For instance, we render the analytic formula $P := \forall A \subseteq \mathbb{N}.\ \mathcal{D}_{\mathsf{a}}\ A \to \mathcal{D}_{\mathsf{a}}\ (\mathbb{N} \setminus A)$ as $[\![P]\!]_{\mathsf{s}} := \forall p{:}\mathbb{N}\to\mathbb{P}.\ \mathcal{D}_{\mathsf{s}}\ p \to \mathcal{D}_{\mathsf{s}}\ (\lambda x{:}\mathbb{N}.\neg p x)$. Here $\mathcal{D}_{\mathsf{a}}$ is the analytic definition of decidability stating that there is a decision function $f : X \to \mathbb{B}$ computable in a fixed model of computation, and $\mathcal{D}_{\mathsf{s}}$ the synthetic rendering stating the existence of any decision function $f : X \to \mathbb{B}$.

We then want our synthetic approach to computability compatible with classical, textbook intuitions to fulfil two central properties. To explain the properties more concretely, we work in a meta-theory strong enough to model CIC and assume an analytic interpretation function $[\![\cdot]\!]_{\mathsf{a}}$ of objects of CIC in the meta-theory. For instance, the function should fulfil $[\![\mathcal{D}_{\mathsf{s}}\ p]\!]_{\mathsf{a}} = \mathcal{D}_{\mathsf{a}}\ [\![p]\!]_{\mathsf{a}}$. Furthermore, we write Ax_{s} for the synthetic computability axioms assumed in CIC.

First, we require the synthetic rendering to be sound: for an analytic proposition P which can be synthetically rendered, we require that $(\vdash_{\mathsf{CIC}} \mathsf{Ax}_{\mathsf{s}} \to [\![P]\!]_{\mathsf{s}}) \to P$. For instance a synthetic theory of computability where all predicates can be proved to be undecidable is not valid, since this directly contradicts a theorem of classical, analytic computability (that there are decidable predicates). Note however that, as is standard in constructive mathematics, we are content with less instances of a definition being provable (there might be more classically, analytically enumerable predicates than constructively synthetically enumerable predicates), and also with less theorems. This means we want that synthetic computability be sound with respect to analytic computability, but not complete.

Requiring soundness is standard in synthetic mathematics. The second property is concerned with axiomatic freedom, and usually not fulfilled. We want our approach to be compatible with classical presentations, in the sense that we want (at least some) axioms used in textbook proofs to be consistently assumable on top of our synthetic axioms. Precisely, we want that $\nvdash_{\mathsf{CIC}} \mathsf{Ax}_{\mathsf{s}} \to \mathsf{LEM} \to \bot$.

We achieve both properties by basing our development on EA, a fully synthetic variant of CT which postulates a *parametrically* universal enumerator. The axiom is inspired by Bauer's axiom, where the set of enumerable sets of natural numbers is assumed enumerable.

Our proofs on paper read similarly to the book of Rogers. The machine-checked proofs have few to no overhead, since all encodings in a model of computation are avoided. Rices' theorem in our synthetic presentation reads virtually the same as textbook formulations, but the formal proof is more elegant and compact, since only the mathematical essence remains.

As further demonstrations of the strengths of this approach, we analyse the construction and theory of both simple and hypersimple sets. As it turns out, no axioms other than EA are necessary, not even Markov's principle, resulting in fully constructive solutions for Post's problem for many-one and truth-table reducibility.

Lastly, we introduce Turing reducibility, show that total bounded Turing reducibility agrees with truth-table reducibility, and that Turing reducibility strictly differs from truth-table reducibility, laying the grounds for future investigations into the Kleene-Post [136] or Friedberg-Muchnik theorems [172, 85].

[172] Muchnik. 1963. On strong and weak reducibility of algorithmic problems.

[85] Friedberg. 1957. Two Recursively enumerable sets of incomparable degrees of unsovlability (solution of Post's problem, 1944.

2.1 Outline

In Chapter 3 we give a short overview of relevant aspects of CIC regarding computability. We discuss constructive proofs, the difference between existential quantification $\exists$ and computational existence Σ, and cover definitions of finite and infinite predicates in CIC.

We introduce synthetic definitions of decidability, enumerability, and semi-decidability in Chapter 4 and prove closure properties and the relation between the notions, all using constructive logic. We intentionally choose synthetic definitions based on total functions, and only in Section 4.5 introduce an abstraction for partial function in CIC, provide an example implementation, and give equivalent definitions of all notions based on partial functions.

In Chapter 5 we introduce one-one, many-one, and truth-table reducibility, and show central properties. All notions of reducibility have in common that they are purely based on total functions. We give elegant characterisations of both truth-table and many-one reducibility in terms of one-one reducibility, with conceptually simpler proofs than the analytic analogues proved e.g. by Rogers [202]. We then give an axiom-free synthetic proof of the Myhill isomorphism theorem, stating that one-one equivalent predicates are isomorphic.

Chapters 6 and 7 can optionally be skipped when reading. Chapter 6 introduces the (analytic) axiom CT, a fully synthetic version SCT, the parametric enumerability axiom EA, an equivalent formulation of SCT. We prove two versions of Rice's theorem based on a variant of SCT and EA to compare the axioms. Chapter 7 surveys the literature regarding the relation of CT and its synthetic variants with axioms like LEM, the axiom of choice, Markov's principle, Weak Kőnig's Lemma, and continuity axioms from Brouwer's intuitionism.

Chapter 8 is a self-contained development of common results in computability theory based on EA. We construct solutions to Post's problem for many-one and truth-table reducibilitity. That means we construct a simple predicate S such that S is enumerable, undecidable, but does not many-one reduce from the halting problem, and a hypersimple predicate H such that H is enumerable, undecidable, but does not truth-table reduce from the halting problem.

Chapter 9 introduces Turing reducibility, a notion of reducibility which is more crucially based on partial functions. We show how total bounded Turing reducibility is exactly truth-table reducibility, and that the hypersimple predicate Turing-reduces to the halting problem.

Appendix A contains inductive types and definitions frequently used and well-known in type theory. Appendix B contains a glossary for the central synthetic notions.

2.2 Related Work

[166] Markov. 1954. The theory of algorithms.

[176] Nagorny. 1995. Andrei markov and mathematical constructivism.

The Russian school of constructivism was founded by Andrei Markov Jr. in 1954 [166]. The school is concerned with constructive developments of mathematical theories, with a particular focus on analysis (see e.g. Nagorny's paper for a high-level overview [176]). What makes the school special is that there is no native notion of numbers or functions. Instead, all of these are defined in terms of words, which can serve as the input to or encode Markov's normal algorithms, a Turing complete model of computability devised by Markov. The only axiom is Markov's principle, which allows proofs by contradiction exactly for termination proofs of algorithms. Switching the perspective, Russian constructivism can be seen as a constructive development of computability theory. It can be seen as both anti-synthetic and anti-analytic: algorithms are the only native object, which are in turn used to model everything else.

Russian constructivism was highly influenced by Kleene's realizability interpretation for constructive logic [133], explicitly hard-wiring realizability. Kleene proves that every total functional relation in Heyting arithmetic corresponds to a μ-recursive function, i.e. the admissibility of *Church's Rule* for Heyting arithmetic, by interpreting all logical formulas as computable functions. In 1965, Kreisel [142] introduces the name CT (*Church's thesis*) for the corresponding axiom, stating that every function is μ-recursive. In fact, he introduces two versions: The axiom CT_0, stating that every function is μ-recursive (we simply call this axiom CT), and the axiom CT_1, stating that every total functional relation is μ-recursive. Troelstra and van Dalen [228] discuss the relation of CT to other axioms. CT plays a prominent role in constructive reverse mathematics, see Diener's [55] overview. It is also used in investigations of the constructiveness of completeness proofs for first-order logic [143]. We are however not aware of any substantial developments of computability theory based on CT.

[142] Kreisel. 1965. Mathematical logic.

[228] Troelstra and van Dalen. 1988. Constructivism in mathematics. Vol. I.

[55] Diener. 2020. Constructive Reverse Mathematics.

Bishop-style constructive analysis [23] can be seen as an implicitly synthetic development of recursive analysis, since every function is considered to be computable rather than explicitly assumed to be. In this framework, Richman [198] identifies a purely synthetic variant of CT allowing to prove results in recursive analysis which contradict classical analysis. The book by Bridges and Richman [25] develops recursive analysis based on Richman's axiom, but does not consider more standard computability theoretic topics such as reducibility or Rice's theorem.

[23] Bishop and Bridges. 2012. Constructive analysis.

[198] Richman. 1983. Church's thesis without tears.

[25] Bridges and Richman. 1987. Varieties of constructive mathematics.

The first steps in synthetic computability theory are due to Bauer [10]. Bauer works in the effective topos, which can be understood as constructive set theory with (1) Markov's principle, (2) dependent and countable choice, and (3) the axiom that the set of enumerable sets of natural numbers is enumerable. The last axiom is equivalent to Richman's axiom, but this is not discussed in detail. As is well-known [228], the assumption of choice and such an enumerability axiom makes the theory anti-classical, i.e. the law of excluded middle is false. Bauer makes advantages of these anti-classical theories and simplifies theorems to their essence. For instance, Rice's theorem (usually stating that non-trivial semantic sets are undecidable) becomes "If A is a set such that all functions of type $A\to A$ have a fixed-point, every function of type $A\to\mathbb{B}$ is constant". The proof of this theorem in Bauer's setting captures indeed exactly the essence of Rice's theorem, but unfortunately the statement of the theorem becomes a classical triviality: If all functions $A\to A$ have a fixed-point, A is a subsingleton. Thus, Bauer's goals can be seen as orthogonal to ours: While Bauer condenses theorems and proofs to their essence, we want to keep theorem statements directly connected to their analytic versions, and only then try to condense the proofs as much as possible.

[10] Bauer. 2006a. First steps in synthetic computability theory.

2.3 Mechanisation in Coq

The Coq code covering the results of this chapter can be found in the following repository:

`https://github.com/uds-psl/coq-synthetic-computability`

The repository contains around 10.000 LoC (25% specification vs. 75% proofs), of which around 1.000 LoC were implemented by Felix Jahn as part of his Bachelor's thesis advised by the author.

CHAPTER 3

Aspects of CIC

We work in the **Calculus of Inductive Constructions (**CIC**)** [39, 181, 182], the computational and constructive type theory underlying the Coq proof assistant [222].

Despite CIC being a historical system which was extended in multiple dimensions over the time, we reserve the term CIC exactly for the type theory Coq implements at the time of writing this thesis, i.e. in version `8.13.2`. For the scope of the thesis, this terminology is not misleading: we believe that all results also hold in the initial, historical version of CIC– but of course do not have formal proof of this claim. The reference manual of Coq provides an overview of CIC as it is currently implemented [223].[1]

[39] Coquand and Huet. 1988. The Calculus of Constructions.

[181] Paulin-Mohring. 1993. Inductive definitions in the system Coq rules and properties.

[182] Paulin-Mohring. 2015. Introduction to the Calculus of Inductive Constructions.

[222] The Coq Development Team. 2021a. The Coq Proof Assistant version 8.13.2.

CIC, as all type theories, is both a logical and a computational system. What makes CIC stand out in comparison to other type theories such as impredicative Martin-Löf type theory (MLTT) or univalent type theory (HoTT). is that there is a syntactic impredicative[2] universe of propositions $\mathbb{P}$, which is separated from the computational universes denoted by $\mathbb{T}_i$ (where we leave out the index i henceforth). Since proposition are types, they behave constructively, meaning e.g. proof by contradiction is not allowed in general. However, Coq fosters an almost maximal axiomatic freedom: Classical axioms cannot be proved, but they can be consistently assumed. This holds for the law of excluded middle and the axiom of choice, but also of weakenings thereof, such as the (weak) limited principle of omniscience, Markov's principle, or unique choice on countable types. In contrast, MLTT proves the axiom of choice, whereas HoTT with a semantic notion of propositions satisfies unique choice.

The separation of the logical universe $\mathbb{P}$ in CIC entails that almost no computational eliminations on proofs $H : P$ where $P : \mathbb{P}$ into a computation $x : X$ where $X : \mathbb{T}$ are allowed. Thus, total functional relations and functions are different objects. While they correspond on the meta-level, i.e. for every relation which can be proved total and functional without assumptions one can also construct a function, this correspondence is not logically available inside the type theory. The correspondence can be made available via an axiom, thereby breaking the separation of propositions and computation: The type-theoretic axiom of (functional) choice yields a function $f : X \to Y$ for every total relation.

In this chapter, we illustrate the interplay between proof/computation and relation/function with three examples: Pigeonhole principles, definitions of finiteness, and definitions of infinitness of predicates. All crucially rely on the constructive interpretation of a $\forall\exists$ quantification in classical logic, which can be rendered as the existence of a function, a $\forall\exists$, or a $\forall\neg\neg\exists$ quantification in type theory, which all have different computational content.

In this first part of the thesis, these variations will play a big role, since we identify functions with computable functions, meaning constructive definitions with computational content will also be implicitly interpreted differently.

[1] To be more precise, CIC is Martin Löf type theory with a hierarchy of type universes [179], with an impredicative universe of propositions as in the calculus of constructions (CoC) [39], inductive types [181] potentially polymorphic in universes [226], template polymorphism, and η-expansion for functions.

[2] i.e. $P : \mathbb{P}$ is a judgement which can be inferred by the type checker, not a proof obligation.

Further details like inductive types and functions on these types we use can be found in Appendix A.

Outline In Section 3.1 we discuss which classical proof rules are available in constructive proofs, whereas in Section 3.2 we illustrate in which restricted cases choice principles are available. In Section 3.3 we prove three pigeonhole principles, illustrating the three forms of forall-exists available in Coq. in Section 3.4 and Section 3.5 we discuss standard constructive definitions of finite and infinite predicates and types.

Publications All sections apart from Section 3.2 contain adapted pieces of text from [72], which were written solely by the author of this thesis.

[72] Forster, Jahn, and Smolka. "A Constructive and Synthetic Theory of Reducibility: Myhill's Isomorphism Theorem and Post's Problem for Many-one and Truth-table Reducibility in Coq." *Pre-print*.

3.1 Constructive proofs

Def. stable proposition

Def. logically decidable proposition

A proposition $P : \mathbb{P}$ is **stable** if it is unchanged under double negation, i.e. $\neg\neg P \to P$. Furthermore, we say that P is **logically decidable**, if $P \vee \neg P$ holds.

Fact 3.1. Logically decidable propositions are stable.

Note that the converse direction is not provable: For an independent proposition P, $\neg P$ is stable but not logically decidable.

Def. law of excluded middle

The **law of excluded middle** LEM states that every proposition is logically decidable.[3]

$$\mathsf{LEM} := \forall P : \mathbb{P}.\ P \vee \neg P$$

LEM is routinely used in many branches of mathematics, often in the form of double negation elimination:

Fact 3.2. $\mathsf{LEM} \leftrightarrow \forall P : \mathbb{P}.\ \neg\neg P \to P$

The converse direction of double negation elimination is however probable:

Fact 3.3. $\forall P : \mathbb{P}.\ P \to \neg\neg P$

It is folklore that LEM is independent in CIC: It can be consistently assumed, but cannot be proved. We will discuss classical logical axioms in more detail in Chapter 7.

In general, a case analysis on a proposition P is possible without LEM if P is logically decidable, but also if the conclusion Q is a stable proposition:

Fact 3.4. If $P \to Q$ and Q is stable, then $\neg\neg P \to Q$.

Corollary 3.5. If Q is stable, then $((P \vee \neg P) \to Q) \to Q$.

Proof. By Fact 3.4, since $\neg\neg(P \vee \neg P)$ is a tautology in constructive logic. ∎

Def. Markov's Principle

Markov's Principle (MP) is a consequence of LEM accepted for instance in Russian constructivism and states that satisfiability of a boolean test on natural numbers is stable:

[40] Coquand and Mannaa. 2017. The Independence of Markov's Principle in Type Theory.

$$\mathsf{MP} := \forall f : \mathbb{N} \to \mathbb{B}.\ \neg\neg(\exists n.\ f\, n = \mathsf{true}) \to (\exists n.\ f\, n = \mathsf{true})$$

MP is also independent in CIC [40] and will be discussed in more detail in Chapter 7. Furthermore, MP is admissible in CIC [184]: Whenever for a defined function f the propositon $\neg\neg(\exists n.\, f\, n = \mathsf{true})$ is provable, one can turn the proof into a proof of $\exists n.\, f\, n = \mathsf{true}$ without use of axioms.

[184] Pédrot and Tabareau. 2018. Failure is Not an Option.

Markov's principle allows case analysis for arbitrary propositions P on goals of the form $Q := \exists n.\, f\, n = \mathsf{true}$:

Fact 3.6. Let MP be given and $Q := \exists n.\, f\, n = \mathsf{true}$ for some $f : \mathbb{N} \to \mathbb{B}$. Then $((P \vee \neg P) \to Q) \to Q$.

In Chapter 7 we will discuss the limited principle of omniscience LPO, which conversely allows case analysis for $P := \exists n.\, f\, n = \mathsf{true}$ and arbitrary Q.

For ease of language we reserve the term "fully constructive" to mean "provable in CIC without any axioms". With this meaning of the word, LEM is not fully constructive, and neither is MP.[4]

3.2 Choice principles

In CIC, both dependent pairs (Σ) and existential quantification $\exists$ can be defined using inductive types. We verbalise $\exists x$ with "there exists x" and in contrast Σx as "one can construct x". Dependent pairs can be eliminated into arbitrary contexts, i.e. there is an elimination function of type

Σ → Sec. A.1, Page 207

$$\forall p : (\Sigma x.Ax) \to \mathbb{T}.\ (\forall x : X.\forall y : Ax.\ p(x,y)) \to \forall s.\ ps.$$

In contrast, existential quantification can only be eliminated for $p : (\exists x.Ax) \to \mathbb{P}$.

This is because CIC forbids so-called large eliminations [181] on the inductively defined $\exists$ predicate. To avoid dealing with Coq's `match`-construct for eliminations in detail, we instead talk about **large elimination principles**. A large elimination principle for $\exists$, which would have the following type, is *not* definable in CIC:

[181] Paulin-Mohring. 1993. Inductive definitions in the system Coq rules and properties.

Def. large elimination principles

$$\forall p : (\exists x.Ax) \to \mathbb{T}.\ (\forall x : X.\forall y : Ax.\ p(x,y)) \to \forall s.\ ps.$$

In particular, this means that one *cannot* define a function of the following type in general

$$\forall p : Y \to \mathbb{P}.\ (\exists y.\ py) \to \Sigma y : Y.\ py.$$

However, such an elimination of $\exists$ into Σ is **admissible** in CIC. This means that any concretely given, fully constructive proof of $\exists y.\ py$ without assumptions can always be given as a proof of $\Sigma y.\ py$. Note that admissibility of a statement is strictly weaker than provability, and in general does not even entail its consistency.

Crucially, CIC allows defining large elimination principles for the falsity proposition $\bot$ and for equality. Additionally, for some restricted types Y and restricted predicates p, one *can* define a large elimination principle for existential quantification. In particular, this holds for $Y = \mathbb{N}$ and $p(n : \mathbb{N}) := f\, n = \mathsf{true}$ for a function $f : \mathbb{N} \to \mathbb{B}$.

[3] Note that all symbols in sf-font like LEM or MP are hyper-linked with their definition in the pdf version of this thesis.

[4] In e.g. the Russian school of constructivism, MP is deemed constructive. We want to emphasise that that our use of the word is a mere simplification of language, not implying any philosophical standpoint considering the constructiveness of MP.

Fact 3.7. One can define functions of type

$\forall A{:}\,\mathbb{T}.\ \bot \to A$

$\forall X{:}\,\mathbb{T}.\forall A{:}\,X \to \mathbb{T}.\forall x_1 x_2{:}\,X.\ x_1 = x_2 \to A x_1 \to A x_2.$

$\forall f{:}\,\mathbb{N} \to \mathbb{B}.\ (\exists n.\ f\,n = \mathsf{true}) \to \Sigma n.\ f\,n = \mathsf{true}$

Corollary 3.8. There is a guarded minimisation function $\mu_{\mathbb{N}}$ of the following type:

$$\mu_{\mathbb{N}} : \forall f{:}\,\mathbb{N} \to \mathbb{B}.\ (\exists n.\ f\,n = \mathsf{true}) \to \Sigma n.\ f\,n = \mathsf{true} \wedge \forall m.\ f\,m = \mathsf{true} \to m \geq n$$

There are two different implementations of such a minimisation function in Coq's Standard Library.[5] One is based on a predicate admitting large elimination like G above, the other proof uses the fact that Coq's type theory allows the definition via `fix` and `match` as opposed to other type theories relying on recursors for types. See e.g. [155, §2.7, §4.1, §4.2] and [20, §14.2.3, §15.4] for a contemporary overview how to implement large eliminations principles.

[38] Coquand. 1989. Metamathematical investigations of a calculus of constructions.

We will not need any other large elimination principle in this thesis. A restriction of large elimination in general is necessary for consistency of Coq [38]. As a by-product, the computational universe $\mathbb{T}$ is separated from the logical universe $\mathbb{P}$, allowing classical logic in $\mathbb{P}$ to be assumed while the computational intuitions for $\mathbb{T}$ remain intact.

The intricate interplay between Σ and $\exists$ is in direct correspondence to the status of the axiom of choice in CIC. The axiom of choice was first stated for set theory by Cantor. In the formulation by Cantor, it is equivalent to the statement that every total, binary relation contains the graph of a function, i.e.:

$$\forall R \subseteq X \times Y.\ (\forall x.\exists y.(x,y) \in R) \to \exists f{:}\,X \to Y.\forall x.\ (x, f\,x) \in R$$

Here $X \to Y$ is the set-theoretic function space. As usual, such a classical principle can also be stated in type theory. However, the concrete formalisation crucially depends on how the notion of a (set-theoretic) function is translated: While in set theory the term *function* is just short for *functional relation*, in CIC functions and (total) functional relations are different objects, we thus discuss both possible translations of the axiom of choice here.

[23] Bishop and Bridges. 2012. Constructive analysis.

The more common version, used e.g. by Bishop and Bridges [23], is to use type-theoretic functions for set-theoretic functions, i.e. state the type-theoretic axiom of (functional) choice as

$$\forall R{:}\,X \to Y \to \mathbb{P}.\ (\forall x.\exists y.\ R x y) \to \exists f{:}\,X \to Y.\forall x.\ R x (f\,x)$$

$\|\cdot\|$ → Sec. A.4, Page 209

Since in type theory proofs are first class object, one can equivalently state

$$\|\forall p{:}\,Y \to \mathbb{P}.\ (\exists y.\ p\,y) \to \Sigma y{:}\,Y.\ p\,y\|$$

Note how this is exactly the non-provable correspondence of $\exists$ and Σ discussed above. This formulation makes clear why in Martin-Löf type theory as implementation of Bishop's constructive mathematics, where one defines $\exists := \Sigma$, the axiom of choice is accepted since it can be proved. In the context of Church's simple type theory, this axiom is also known as axiom of indefinite description [3].

[3] Andrews. 2002. An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof.

[5] The idea was conceived independently by Benjamin Werner and Jean-François Monin in the 1990s.

A third, simpler characterisation is possible and used in the book on Homotopy Type Theory [231]:

$$\forall X:\mathbb{T}.\ \forall P: X \to \mathbb{T}.\ (\forall x.\ \|Px\|) \to \|(\|\forall x.\ Px)$$

A variant of the axiom of choice is the axiom of unique choice:

∃! → Sec. A.2, Page 208

$$\forall R: X \to Y \to \mathbb{P}.\ (\forall x.\exists! y.\ Rxy) \to \exists f: X \to Y.\ Rx(fx)$$

The axiom of unique choice holds in homotopy type theory [231].

Note that the condition implies that R is total and functional. Thus, using the notation $R: X \rightsquigarrow Y$ for functional relations, the axiom of unique choice can also be stated as

$$\forall R: X \rightsquigarrow Y.\ (\forall x.\exists y.\ Rxy) \to \exists f: X \to Y.\ Rx(fx)$$

Again, in type theory one can equivalently state

$$\|\forall p: Y \to \mathbb{P}.\ (\exists y!.\ py) \to \Sigma y: Y.\ py\|$$

In the context of Church's simple type theory, this axiom is also known as axiom of definite description [16].

Due to the separation of $\mathbb{P}$ from the computational universes $\mathbb{T}$, neither of the two variants of the axiom of choice are provable in CIC. There are however two notable classes of choice principles which are provable: First, choice principles built on top of the guarded minimisation operator $\mu_{\mathbb{N}}$ from Corollary 3.8. Second, when X is a finite discrete type, choice principles are provable, see Lemma 3.29. We discuss all provable choice principles in Section 7.6.1. In Section 7.6 we more generally discuss the axiom of choice, the axiom of countable choice, and the axiom of dependent choice.

3.3 Pigeonhole principles

Similar to choice principles, pigeonhole principles are omnipresent in discrete mathematics. We prove three constructive variants of pigeonhole principles, based on duplicate-free lists: Our formulation of the pigeonhole principle is that for any duplicate-free list l_1 longer than a list l_2 one can obtain an element x which is in l_1 but not in l_2 – for different formalisations of "obtain" in CIC.[6]

Precisely, we will prove principles of the following forms:

1. x is computable: $\forall l_1 l_2.\ \cdots \to \Sigma x.\ \ldots$
2. x constructively exists: $\forall l_1 l_2.\ \cdots \to \exists x.\ \ldots$
3. x classically exists: $\forall l_1 l_2.\ \cdots \to \neg\neg\exists x.\ \ldots$

These are exactly the three possible formalisations of "obtain" in CIC. A function returning a dependent pair (Σ), a proof of an existential proposition ($\exists$), or of a double-negated existential proposition ($\neg\neg\exists$, equivalently $\neg\neg\Sigma$).

[6] Admittedly, this formulation of the pigeonhole principles is more a "pigeon-less hole principle": Instead of proving that if there are more pigeons than holes, then two pigeons end up in the same hole, we show that if there are more holes than pigeons, then there is hole which stays empty.

Formulating existence as Σ inherently means that the result has to be *computable*, a property unchanged by the assumption of logical axioms like LEM. In the absence of axioms, $\exists$ has to be proved using a (computable) function, but the function cannot be used computationally after the proof.

We now turn towards proving the principles, which will vary in the requirements on the underlying type X. For the formalisation, we define $\#l : \mathbb{P}$ for a list $l : \mathbb{L}X$ inductively to state that l does not contain any duplicates:

$$\frac{}{\#[\,]} \qquad\qquad \frac{x \notin l \qquad \#l}{\#(x :: l)}$$

The $\forall\Sigma$-version of the pigeonhole principle is straightforward given a computational equality decider for X. We model a computational equality decider X for now via a dependently typed function of type $\forall x_1 x_2 : X.\ (x_1 = x_2) + (x_1 \neq x_2)$.

+ on types
→ Sec. A.1, Page 207

Lemma 3.9. Let $d : \forall x_1 x_2 : X.(x_1 = x_2) + (x_1 \neq x_2)$, and $l_1, l_2 : \mathbb{L}X$. If $\#l_1$ and $|l_1| > |l_2|$, then $\Sigma x.\ x \in l_1 \wedge x \notin l_2$.

Proof. By induction on the derivation of $\#l_1$, with l_2 generalised.

The case $l_1 = [\,]$ is contradictory, since $|[\,]| = 0 > |l_2|$ is impossible.

Let $x \notin l_1$ and $\#l_1$. Case analysis on $(x \in l_2) + (x \notin l_2)$, possible using d. If $x \notin l_2$, the claim is immediate. If $x \in l_2$, the claim follows from the induction hypothesis for $l_2 := \mathsf{filter}(\lambda y.\neg_{\mathbb{B}}(d\,x\,y))l_2$ (i.e. for l_2 with x removed). ∎

Note that the proof could be adapted such that a function $d : \forall x\,y : X.(x \neq y) + (\neg x \neq y)$ would also suffice. The proof could alternatively be given in form of a function of type $\mathbb{L}X \to \mathbb{L}X \to \mathbb{O}X$ which returns the element x specified by Σ provided $\#l_1$ and $|l_1| > |l_2|$.

Note how thus this $\forall\Sigma$ version crucially depends on computationally removing an element from a list via the equality decider d. If no such d is available, a removal function is not definable. However, for the $\forall\exists$ and $\forall\neg\neg\exists$ forms of the pigeonhole principle, a removal *function* is not needed.

Instead, for the $\forall\exists$ version it suffices to prove that for any list l_0 and any element x_0, there exists ($\exists$) a list with the same elements of l_0, just x_0 removed. This becomes possible provided $x_1 \neq x_2$ is logically decidable for all x_1, x_2, i.e. we assume $\forall x\,y : X.\ (x \neq y) \vee (\neg x \neq y)$.

For the $\forall\neg\neg\exists$ version of the pigeonhole principle, consequently a removal principle of the form $\neg\neg\exists l \ldots$ suffices, which can be proved fully constructively without assumptions: $\forall x\,y : X.\ \neg\neg((x = y) \vee (x \neq y))$ is provable for any X.

To prove the two removal principles, we define a generalised filter predicate $l_0 \supseteq_p l$ w.r.t. a predicate $p : X \to \mathbb{P}$ stating that l is exactly the sublist of l_0 with all elements which fulfil p:

$$\frac{}{[\,] \supseteq_p [\,]} \qquad \frac{p\,x \qquad l_0 \supseteq_p l}{(x :: l_0) \supseteq_p (x :: l)} \qquad \frac{\neg p\,x \qquad l_0 \supseteq_p l}{(x :: l_0) \supseteq_p l}$$

Fact 3.10. Let $l_0 \supseteq_p l$. Then the following hold:

1. $x \in l \leftrightarrow x \in l_0 \wedge p\,x$
2. $|l| \leq |l_0|$
3. $\forall x \in l.\ \neg p\,x \to |l| < |l_0|$

We can prove two existence principles, one assuming that p is logically decidable, and one proving a double negation.

Fact 3.11. Let $l_0 : \mathbb{L}X$ and $x_0 : X$.

1. $(\forall x.px \vee \neg px) \to \exists l.\, l_0 \supseteq_p l$
2. $\neg\neg\exists l.\, l_0 \supseteq_p l$

Using the two removal principles, we can immediately prove the $\forall\exists$ and $\forall\neg\neg\exists$ forms of the pigeonhole principle.

Lemma 3.12. If $\#l_1$ and $|l_1| > |l_2|$, then $\neg\neg\exists x.x \in l_1 \wedge x \notin l_2$.

Proof. By induction on the derivation of $\#l_1$, with l_2 generalised.

The case $l_1 = [\,]$ is contradictory, since $|[\,]| = 0 > |l|_2$ is impossible.

Let $x \notin l_1$ and $\#l_1$. Since the claim is negative, we can do a case analysis on $x \in l_2$ by Corollary 3.5. If $x \notin l_2$, the claim is immediate. If $x \in l_2$, we obtain l s.t. $l_2 \supseteq_{(\lambda y.x \neq y)} l$ from Fact 3.10 (2). The claim follows by induction for l. ∎

Lemma 3.13. If $\forall x_1 x_2 : X.x_1 \neq x_2 \vee \neg x_1 \neq x_2$, $\#l_1$ and $|l_1| > |l_2|$, then $\exists x.\, x \in l_1 \wedge x \notin l_2$.

Proof. Similar to the last proof. We use the case analysis $x \notin l_2 \vee \neg x \notin l_2$, possible by the assumption that non-equality is logically decidable, and Fact 3.10 (1) with $py := x \neq y$. ∎

3.4 Finite types and predicates

The most straightforward definition of finite types in CIC is to ask for a list $l : \mathbb{L}X$ such that $\forall x : X.\, x \in l$. This notion can be generalised to predicates $p : X \to \mathbb{P}$, and it is then possible to recover the notion of finite types by instantiating to the full predicate $\lambda x.\, \top$.

For predicates, several notions based on lists are possible as definition of finiteness. This is well-known in conventional presentations of constructive mathematics, where one talks about (Bishop-)finite ($\mathcal{B}$-finite) and subfinite ($\widetilde{\mathcal{B}}$-finite) sets.

We discuss both notions of finiteness in the setting of CIC, and for conciseness call them finiteness and subfiniteness. Note however that the use of the word finite then differs from more usual formalisations of Bishop style constructive mathematics, e.g. to predicative Martin-Löf type theory (MLTT) as implemented in the proof assistant Agda [177]. In CIC, equality on finite predicates and types is not necessarily logically decidable, whereas in predicative MLTT it is even computationally decidable [68]. However, also in CIC the equivalence of finiteness and subfiniteness is equivalent to the law of excluded middle.

[68] Firsov and Uustalu. 2015. Dependently typed programming with finite sets.

Since finite types can be recovered from finite predicates we start with the discussion of predicates.

A list $l : \mathbb{L}X$ **lists** a predicate $p : X \to \mathbb{P}$ if $\forall x : X.\, px \leftrightarrow x \in l$. A predicate p is **finite** if there exists such a list, i.e.

Def. finite predicate

$$\mathcal{F}p := \exists l : \mathbb{L}X.\forall x : X.\, px \leftrightarrow x \in l$$

Note that the predicate $\lambda x.\, \bot$ is only listed by $[\,]$ and no other list, while the predicate $\lambda x.\, x = x_0$ is listed by $[x_0]$, $[x_0, x_0]$ and so on - but the list is not allowed to contain elements not fulfilled by a predicate. To ensure finiteness, this is unnecessarily strict, and we can relax the condition.

Def. subfinite predicate

A list $l:\mathbb{L}X$ **exhausts** a predicate $p:X\to\mathbb{P}$ of $\forall x:X.\ px\to x\in l$. A predicate $p:X\to\mathbb{P}$ is **subfinite** if there exists an exhausting list, i.e.:

$$\mathcal{X}p := \exists l:\mathbb{L}X.\ \forall x:X.\ px\to x\in l$$

$\mathbb{1}$ → Sec. A.1, Page 207

Clearly, listability implies subfiniteness, but the converse implication is equivalent to excluded middle – even when considered just on the unit type $\mathbb{1}$:

Fact 3.14. If l lists p, l exhausts p.

Corollary 3.15. Finite predicates are subfinite.

Lemma 3.16. Every exhaustible predicate is not not finite, i.e. $\forall p.\ \mathcal{X}p\to\neg\neg\mathcal{F}p$.

Proof. We first prove the general lemma

$$\forall l_0:\mathbb{L}X.\forall p:X\to\mathbb{P}.\neg\neg\exists l'.\forall x.\ x\in l'\leftrightarrow x\in l_0\wedge px$$

by induction on l_0.

Let l exhaust p and let p be not finite. We have to prove falsity. Using the lemma, we obtain l' s.t.$\forall x.\ x\in l'\leftrightarrow x\in l_0\wedge px$ and still have to prove falsity. Now l' lists p. Contradiction. ∎

Lemma 3.17. If every subfinite predicate is finite, the law of excluded middle holds.

Proof. Let every subfinite predicate be finite and $P:\mathbb{P}$. We define $p(x:\mathbb{1}) := P$. p is finite because it is exhausted by $[\star]$. If p is listed by l, case analysis on l allows proving $P\vee\neg P$. ∎

Corollary 3.18. Every subfinite predicate is finite if and only if excluded middle holds.

We conclude by establishing some predicates as finite and with useful closure properties.

Fact 3.19. Let $n_1,n_2:\mathbb{N}$ and $x_0:X$. The following are all finite:

1. $\lambda x.\ x=x_0$,
2. $\lambda n.\ n_1\le n\le n_2$,
3. $\lambda n.\ n_1\le n<n_2$.
4. $\lambda l:\mathbb{L}\mathbb{B}.\ |l|=n_1$, and
5. $\lambda l:\mathbb{L}\mathbb{B}.\ |l|<n_1$.

Lemma 3.20. Let $p,q:X\to\mathbb{P}$ be sub-finite, and $r:X\to\mathbb{P}$ be any predicate.

1. $\lambda x.\ px\wedge rx$ is sub-finite.
2. $\lambda x.\ px\vee qx$ is sub-finite.

Proof. For (1), if l exhausts p, l exhausts $\lambda x.\ px\wedge rx$. For (2), if l_1 exhausts p and l_2 exhausts q, $l_1+\!\!+\,l_2$ exhausts $\lambda x.\ px\vee qx$. ∎

Lemma 3.21. Let p and q be finite, and r any predicate.

1. $\lambda x.\ px\wedge rx$ is sub-finite.
2. $\lambda x.\ px\vee qx$ is finite.

Proof. For (1) if p is finite it is sub-finite by Corollary 3.15, thus $\lambda x.\ px\wedge rx$ is exhaustible by Lemma 3.20 (1). For (2), if l_1 lists p and l_2 lists q, $l_1+\!\!+\,l_2$ lists $\lambda x.\ px\vee qx$. ∎

3.4.1 Finite types

A list $l : \mathbb{L}X$ **lists** a type X if $\forall x{:}X.\ x \in l$. A type X is **finite** if there exists such a list, i.e.: **Def.** finite type

$$\mathcal{F}X := \exists l{:}\mathbb{L}X.\ \forall x{:}X.\ x \in l$$

As mentioned, the definition of finite types is equivalent to instantiating both finiteness and subfiniteness to the full predicate:

Fact 3.22. The following are equivalent:

1. X is finite.
2. $\lambda x{:}X.\ \top$ is subfinite.
3. $\lambda x{:}X.\ \top$ is finite.

Canonical finite types arise from the type family $\mathbb{F} : \mathbb{N} \to \mathbb{T}$, defined inductively with two constructors as follows:

$$\frac{}{\mathsf{O} : \mathbb{F}_{\mathsf{S}\,n}} \qquad\qquad \frac{i : \mathbb{F}_n}{\mathsf{S}\,i : \mathbb{F}_{\mathsf{S}\,n}}$$

Fact 3.23. $\mathbb{1}$, $\mathbb{B}$, and $\mathbb{F}_n$ are finite types.

Proof. $\mathbb{1}$ is listed by $[\star]$, $\mathbb{B}$ is listed by $[\mathsf{true}, \mathsf{false}]$, and one can construct a list l_n listing $\mathbb{F}_n$ by recursion on n. ∎

Fact 3.24. Finite types are closed under pairs, sums, options, and vectors of length n.

Proof. If l_X lists X and l_Y lists Y,

1. $l_X \times l_Y$ lists $X \times Y$.
2. $[\mathsf{inl}\, x \mid x \in l_X] +\!\!+ [\mathsf{inr}\, y \mid y \in l_Y]$ lists $X + Y$.
3. $\mathsf{None} :: [\mathsf{Some}\, x \mid x \in l_X]$ lists $\mathbb{O}X$
4. $l_0 := [\,[\,]\,]$ lists X^0, and $l_{\mathsf{S}\,n} := [x :: v \mid (x, v) \in l_X \times l_n]$ lists $X^{\mathsf{S}\,n}$. ∎

Corollary 3.25. If X is finite and there is a function $d : \forall x\, y : X.(x = y) + (x \neq y)$, then there exists $n : \mathbb{N}$ such that X is isomorphic to $\mathbb{F}_n$. → Sec. A.4, Page 209

Finite types have special properties. Every predicate on a finite type is sub-finite.

Fact 3.26. If X is finite and $p : X \to \mathbb{P}$, then p is sub-finite.

A finite choice principle is provable for all total relations. The easiest proof of the principle is obtained for the following formulation.

Lemma 3.27. Let $L : \mathbb{L}X$, $R : X \to Y \to \mathbb{P}$, and $\forall x \in L.\exists y.\, Rx\,y$. Then $\exists L'.\ \mathsf{Forall}_2\ R\ L\ L'$.

Proof. By induction on L. ∎

The principle can be more conveniently used if there instead is a function returning the corresponding elements for L, which requires X to have decidable equality.

Lemma 3.28. Let X, Y be types, $y_0 : Y$, $D{:}\forall xy{:}X.\,(x = y) + (x \neq y)$, $R{:}X \to Y \to \mathbb{P}$, $p{:}X \to \mathbb{P}$ be finite, and $\forall x.px \to \exists y.\,Rxy$, then $\exists f : X \to Y.\forall x.\,px \to Rx(fx)$.

Proof. Let L list p. By induction on L. If $L = []$, pick $\lambda x.y_0$. If $L = x' :: L$ we have y' s.t. $Rx'y'$ by assumption. Let f be the function from the inductive hypothesis and pick $\lambda x.$ **if** Dxx' **then** y' **else** fx'. ∎

Lemma 3.29. If X is finite, $D{:}\forall xy.\,(x = y) + (x \neq y)$, $R{:}X \to Y \to \mathbb{P}$, and $\forall x.\exists y.\,Rxy$, then $\exists f : X \to Y.\forall x.\,Rx(fx)$.

Proof. Let L list X. If $L = []$, f can be defined using computational explosion. If L contains x_0 one can obtain $y_0 : Y$ and use Lemma 3.28. ∎

3.5 Infinite Predicates

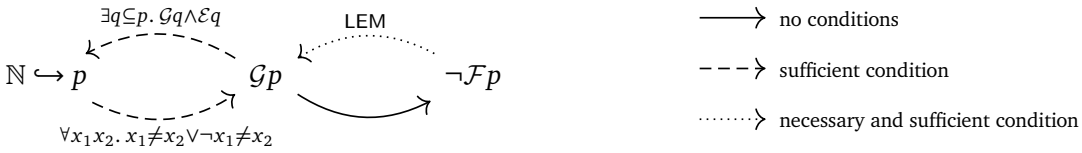
Similar to finite predicates, there are several classically equivalent but constructively different definitions of infinite predicates. The different notions become especially interesting when analysed under the lens of synthetic computability theory, i.e. by interpreting the function space as computable functions.

We discuss three possible definitions of infinity: Non-finiteness, generativity, and Cantor infinity. A predicate $p{:}X \to \mathbb{P}$ is

1. non-finite if $\neg\mathcal{F}p$ holds – a stable proposition.
2. generative ($\mathcal{G}p$) if for every list there exists an element satisfying p not in the list – a $\forall\exists$ proposition.
3. Cantor-infinite ($\mathbb{N} \hookrightarrow p$) if there exists an injection of type $\mathbb{N} \to X$ returning only elements in p – equivalent to a $\|\forall\Sigma\|$ proposition.

$\mathcal{E}q$ → Sec. 4.2, Page 33

We obtain the following graph:



We now turn to the formal definitions and proofs.

Def. non-finite predicate

A predicate $p{:}X \to \mathbb{P}$ is called **non-finite** if $\neg\mathcal{F}p$ holds, i.e. if

$$\neg\exists l.\forall x.\,px \leftrightarrow x \in l$$

$\mathcal{F}p$ → Sec. 3.4, Page 25

Lemma 3.30. A predicate is non-finite if and only if $\neg\mathcal{X}p$ holds, i.e. $\neg\mathcal{F}p \leftrightarrow \neg\mathcal{X}p$.

$\mathcal{X}p$ → Sec. 3.4, Page 25

Proof. $\mathcal{X}p \to \neg\neg\mathcal{F}p$ proved in Lemma 3.16 implies the direction from left to right.

$\mathcal{F}p \to \mathcal{X}p$ proved in Corollary 3.15 implies the converse direction. ∎

Def. generative predicate

A predicate $p{:}X \to \mathbb{P}$ is called **generative** if for every list one can find an element satisfying p which is not in the list:

$$\mathcal{G}p := \forall l{:}\mathbb{L}X.\exists x.\,px \wedge x \notin l$$

Fact 3.31. Generative predicates are inhabited, and non-finite predicates are not not inhabited.

Generative predicates on $\mathbb{N}$ can be characterised as follows:

Fact 3.32. $\forall p\colon \mathbb{N} \to \mathbb{P}.\ \mathcal{G}p \leftrightarrow \forall n.\exists m \geq n.\ pm$

The following characterisation lemma of non-finiteness as $\forall\neg\neg\exists$-form of generativity allows proving the connection of the two notions:

Lemma 3.33. If $\forall x_1 x_1\colon X.\ x_1 = x_1 \vee x_1 \neq x_2$, then $\neg\mathcal{X}p \leftrightarrow \forall l\colon \mathbb{L}X.\neg\neg\exists x.px \wedge x \notin l$.

Proof. The direction from right to left is straightforward.

For the direction from left to right, let equality on X be logically decidable and p be non-finite. Let $l\colon \mathbb{L}X$. We have to prove that $\neg\neg\exists x.\ px \wedge x \notin l$. We assume $\neg\exists x.\ px \wedge x \notin l$ and derive a contradiction.

We do so by proving that p is exhausted by l: Let x s.t. px. We have to show $x \in l$. Since equality on X is logically decidable, $x \in l$ is logically decidable. It thus suffices to assume $x \notin l$ and derive a contradiction.

But now $px \wedge x \notin l$ holds. Contradiction. ∎

Thus, on types with logically decidable equality, and in particular on discrete types, non-finiteness is the $\forall\neg\neg\exists$ version of generativity.

Corollary 3.34. $\forall p\colon \mathbb{N} \to \mathbb{P}.\ \neg\mathcal{F}p \leftrightarrow \forall n.\neg\neg\exists m \geq n.pm$

The direction from left to right did not depend on logical decidability of equality, we thus can prove:

Fact 3.35. Generative predicates are non-finite.

Lemma 3.36. If LEM holds and p is non-finite, p is generative.

LEM → Sec. 3.1, Page 20

Proof. By Lemma 3.33 since under LEM equality on any type is logically decidable. ∎

We introduce a second $\forall\exists$-notion of infinite predicates: A predicate $p\colon X \to \mathbb{P}$ is **unbounded** if there exist duplicate-free lists of arbitrary length with elements from p:

Def. unbounded predicate

$$\mathcal{U}p := \forall n\colon \mathbb{N}.\exists l.\ |l| = n \wedge \#l \wedge \forall x \in l.\ px$$

Fact 3.37. $\neg\mathcal{F}p \leftrightarrow \forall n\colon \mathbb{N}.\neg\neg\exists l.\ |l| = n \wedge \#l \wedge \forall x \in l.\ px$

Lemma 3.38. Generative predicates are unbounded.

Proof. Let p be generative and $n\colon \mathbb{N}$. We construct l by induction on n. For $n = 0$ we pick $l = [\,]$. For $n = \mathsf{S}\,n'$ we use the inductive hypothesis to obtain l, and use generativity to obtain x s.t. px and $x \notin l$. Now $x :: l$ is the wanted list. ∎

We then can prove the following using Lemmas 3.12 and 3.13.

Fact 3.39. Unbounded predicates are non-finite.

Fact 3.40. Unbounded predicates are generative for types X s.t. $\forall x_1 x_2\colon X.\ x_1 \neq x_2 \vee \neg x_1 \neq x_2$.

LEM is necessary for non-finite predicates to be unbounded:

Lemma 3.41. LEM if all non-finite $p:\mathbb{N}\to\mathbb{P}$ are unbounded.

Proof. Let $P:\mathbb{P}$ and $pn := P \leftrightarrow n \text{ is even}$. If p is inhabited, LEM holds: Given pn we can analysing whether n is even.

Since unbounded predicates are inhabited, by assumption it suffices to prove that p is non-finite. So let p be subfinite, we have to prove falsity. We can thus decide P and it suffices to prove that p is generative to obtain a contradiction.

If P holds, $pn \leftrightarrow n \text{ is even}$, which is generative. If $\neg P$ holds, $pn \leftrightarrow n \text{ is odd}$, which is generative as well. ∎

Corollary 3.42. Non-finite predicates are unbounded if and only if LEM holds.

Corollary 3.43. Non-finite predicates are generative if and only if LEM holds.

Def. Cantor-infinite predicate

Lastly, we introduce Cantor infinity, often used in classical presentations of mathematics and computability theory. A predicate $p:X\to\mathbb{P}$ is **Cantor-infinite** if there exists an injection $f:\mathbb{N}\to X$ only returning elements in p:

$$\mathbb{N}\hookrightarrow p := \exists f:\mathbb{N}\to X.\forall n_1.\, p(f\,n_1)\wedge\forall n_2.\, f\,n_1 = f\,n_2 \to n_1 = n_2$$

Lemma 3.44. Cantor-infinite predicates are unbounded.

Proof. Let $\mathbb{N}\hookrightarrow p$ via f. For n, $[f\,0,\ldots,f\,n]$ proves $\mathcal{U}p$. ∎

Cantor-infinity is the $\forall\Sigma$ version of generativity:

Lemma 3.45. Let X be discrete and $p:X\to\mathbb{P}$. Then

$$\mathbb{N}\hookrightarrow p \leftrightarrow \forall l:\mathbb{L}X.\Sigma x.\, px\wedge x\notin l.$$

Proof. The direction from left to right is a direct adaption of Lemma 3.44 using Lemma 3.9. Conversely, let $F:\forall l{:}\mathbb{L}X.\Sigma x.x\notin l\wedge px$ and $f\,l := \pi_1(Fl)$. Let $g\,0 := []$ and $g(\mathrm{S}\,n) := gn +\!\!+ [f(gn)]$. Then $\lambda n.f(gn)$ proves $\mathbb{N}\hookrightarrow p$. ∎

CHAPTER 4

Decidability and enumerability

In this chapter we give a concise synthetic presentation of decidability and recursive enumerability in CIC. In textbook presentations, these notions are defined as properties of sets of natural numbers. We here define them as properties of predicates $X \to \mathbb{P}$ over arbitrary types X. Thus, we can directly say that a predicate over e.g. lists of numbers is decidable, without having to define Gödel codes for lists.

Most results about the decidability or enumerability of sets of natural numbers do not actually rely on the full structure of $\mathbb{N}$. Sometimes discreteness of $\mathbb{N}$ (i.e. decidability of equality) is needed, sometimes the fact that $\mathbb{N}$ can be enumerated, sometimes infinity of $\mathbb{N}$, and sometimes the fact that $\mathbb{N}$ has an order structure on which to perform recursion. We always state the weakest possible assumption on types for our results.

For instance, over arbitrary types, the well-known equivalent presentations of recursive enumerability as the domain or co-domain of a partial function become non-equivalent. We thus distinguish the notions and present them based on total functions as enumerability (focussing on the co-domain) and semi-decidability (focussing on the domain).

We define all notions using total functions, which simplifies proofs. However, we also define an interface for partial functions in CIC, provide a reference implementation, and prove that all notions can also be defined in terms of such partial functions instead of total functions as sanity check of the definitions.

Outline In Section 4.1 we introduce synthetic decidability and prove closure properties. Section 4.2 introduces synthetic enumerability. We prove most closure properties in Section 4.3, where list enumerability is introduced as equivalent but easier to treat notion of enumerability. Section 4.4 introduces semi-decidability. Section 4.5 introduces synthetic partial functions. All definitions are crucially placed in the universe $\mathbb{P}$, as discussed in Section 4.6.

Publications The results in this chapter are based on

[73] Forster, Kirst, and Smolka. "On synthetic undecidability in Coq, with an application to the Entscheidungsproblem." *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 2019.

[70] Forster. "Church's thesis and related axioms in Coq's type theory." *29th EACSL Annual Conference on Computer Science Logic (CSL 2021)*.

4.1 Decidability

Def. decider

A function $f : X \to \mathbb{B}$ is a **decider** for $p : X \to \mathbb{P}$ if $\forall x : X.\ px \leftrightarrow f\,x = \mathsf{true}$. If f is a decider for p, we also say that f decides p.

Def. decidable predicate

A predicate is **decidable** if there exists a decider:

$$\mathcal{D}(p : X \to \mathbb{P}) := \exists f : X \to \mathbb{B}.\forall x.\ px \leftrightarrow f\,x = \mathsf{true}$$

We extend the notion of decidability defined on unary predicates to n-ary predicates, i.e. deciders of predicates $p: X_1 \to \cdots \to X_n$ are functions $f: X_1 \to \cdots \to X_n \to \mathbb{B}$.

$\equiv$ → Sec. A.3, Page 208

Fact 4.1. Let $f \equiv_{X\to\mathbb{B}} g$, $p \equiv_{X\to\mathbb{P}} q$, and $p', q': X \to \mathbb{P}$.

1. f decides p if and only if g decides q.
2. p is decidable if and only if q is decidable: $\mathcal{D}p \leftrightarrow \mathcal{D}q$.
3. If f decides p' and g decides q', then $p' \equiv_{X\to\mathbb{P}} q'$.

Def. dependently typed decider

In CIC, deciders can equivalently be defined using dependent types. A **dependently typed decider** for a predicate $p: X \to \mathbb{P}$ is a function $f: \forall x{:}X.(px) + (\neg px)$.

Fact 4.2. The following hold:

1. If f decides p one can construct a dependently typed decider for p.
2. If f is a dependently typed decider for p, $\lambda x.$**if** $f\,x$ **is** $\mathsf{inl}\,H$ **then** true **else** false decides p.

if ... **is** → Sec. A.1, Page 207

$\|\cdot\|$ → Sec. A.4, Page 209

Corollary 4.3. Predicates p are decidable if and only if $\|\forall x.\,(px) + (\neg px)\|$.

We will prove all statements in this chapter without making use of dependent deciders, since boolean functions are easier to write out in detail than dependently typed functions returning proofs. For the Coq mechanisation, the dependently typed versions however can be advantageous because they are easier to construct in proof scripts.

We now turn to closure properties of decidable predicates. We first state the constructions for deciders regarding complement, and pointwise conjunction and disjunction:

Lemma 4.4. Let $f: X \to \mathbb{B}$ decide p and $g: X \to \mathbb{B}$ decide q. Then

$\overline{p}$ → Sec. A.4, Page 209

1. $\lambda x.\ \neg_{\mathbb{B}}(f\,x)$ decides $\overline{p}$.
2. $\lambda x.\ f\,x \wedge_{\mathbb{B}} g\,x$ decides $\lambda x.\ px \wedge qx$.
3. $\lambda x.\ f\,x \vee_{\mathbb{B}} g\,x$ decides $\lambda x.\ px \vee qx$.

The following corollary is then immediate:

Corollary 4.5. Decidable predicates are closed under complement, pointwise conjunction, and pointwise disjunction.

Finally, we can construct choice functions for total relations into $\mathbb{N}$ given a decider:

Lemma 4.6. Let $R: X \to \mathbb{N} \to \mathbb{P}$ be total, i.e. $\forall x.\exists n.\ Rxn$, and g be a decider for R. Then one can construct $f: X \to \mathbb{N}$ such that $\forall x.\ Rx(f\,x)$.

Proof. We first prove $\forall x.\exists y.g\,x\,y = \mathsf{true}$ by exploiting totality of R. Let H be this proof. Then we can define $f\,x := \pi_1(\mu_{\mathbb{N}}(Hx))$, which clearly fulfils $\forall x.\ Rx(f\,x)$. ∎

$\mu_{\mathbb{N}}$ → Sec. 3.2, Page 22

4.1.1 Discrete types

Def. equality decider

Def. discrete

A function $f: X \to X \to \mathbb{N}$ is an **equality decider** for a type X if $\forall x_1 x_2{:}X.\ x_1 = x_2 \leftrightarrow f\,x_1 x_2 = \mathsf{true}$. A type X is **discrete** if there exists an equality decider, i.e. if $\mathcal{D}(\lambda x_1 x_2{:}X.x_1 = x_2)$.

Fact 4.7. $\lambda b_1 b_2{:}\mathbb{B}.$ **if** b_1 **then** b_2 **else** $\neg_{\mathbb{B}} b_2$ is an equality decider for $\mathbb{B}$ and one can construct an equality decider for $\mathbb{N}$.

Fact 4.8. Let f and g be equality deciders for X and Y respectively. Then one can construct equality deciders for $X \times Y$, $X + Y$, $\mathbb{O}X$, and $\mathbb{L}X$.

Corollary 4.9. $\mathbb{B}$ and $\mathbb{N}$ are discrete, and discrete types are closed under pairs, sums, options, and lists.

Fact 4.10. If $l : \mathbb{L}X$ lists p and f decides equality on X, then $\lambda x.\ x \in^f_{\mathbb{B}} l$ decides p.

$\in^f_{\mathbb{B}}$ → Sec. A.1, Page 207

Corollary 4.11. Finite predicates on discrete types are decidable.

4.2 Enumerability

Rogers [202] defines a predicate p to be enumerable if it is either empty or the range of a total function. The built-in case distinction requires omnipresent use of classical logic to establish enumerability, we thus use the option type instead:

[202] Rogers. 1987. Theory of Recursive Functions and Effective Computability.

→ Sec. A.1, Page 207

A function $f : \mathbb{N} \to \mathbb{O}X$ is an **enumerator** for $p : X \to \mathbb{P}$ if $\forall x.\ px \leftrightarrow \exists n.\ f\,n = \mathsf{Some}\,x$. We also say that f enumerates p.

Def. enumerator

A predicate is **enumerable** if there exists an enumerator:

Def. enumerable predicate

$$\mathcal{E}(p : X \to \mathbb{P}) := \exists f : \mathbb{N} \to \mathbb{O}X.\forall x.\ px \leftrightarrow \exists n.\ f\,n = \mathsf{Some}\,x$$

Note that with this definition, the empty predicate is enumerated by $\lambda x.\,\mathsf{None}$.

To lift enumerability to n-ary predicates we use (implicit) uncurrying: We call a predicate $p : X_1 \to \cdots \to X_n \to \mathbb{P}$ enumerable if its uncurrying $\lambda(x_1, \ldots, x_n).px_1 \ldots x_n$ is enumerable.

Classically, a predicate is called co-enumerable if $\mathcal{E}\overline{p}$ holds. We refrain from using the terminology to avoid clashes with the constructively stronger notion of co-semi-decidability we introduce later.

We call predicates where both $\mathcal{E}p$ and $\mathcal{E}\overline{p}$ holds **bi-enumerable**.

Def. bi-enumerable

Fact 4.12. Let $f \equiv_{\mathsf{ran}} g$, $p \equiv_{X\to\mathbb{P}} q$, and $p', q' : X \to \mathbb{P}$.

$\equiv$ → Sec. A.3, Page 208

1. f enumerates p if and only if g enumerates q.
2. p is enumerable if and only if q is enumerable: $\mathcal{E}p \leftrightarrow q$.
3. If f enumerates p' and g enumerates q', then $p' \equiv_{X\to\mathbb{P}} q'$.

Similarly to discrete types, the notion of enumerability can be lifted to the type level as well. A function $f : \mathbb{N} \to \mathbb{O}X$ is an **enumerator for a type** X if $\forall x.\exists n.f\,n = \mathsf{Some}\,x$, and we call such types **enumerable type**s.

Def. enumerator for a type

Def. enumerable type

Lemma 4.13. If f decides p and e enumerates X, then one can construct an enumerator for p.

Proof. Take $\lambda n.$**if** en **is** $\mathsf{Some}\,x$ **then if** $f\,x = \mathsf{true}$ **then** $\mathsf{Some}\,x$ **else** None **else** None. ∎

Corollary 4.14. If p is decidable and X is enumerable then p and $\overline{p}$ are enumerable.

Lemma 4.15. Let f enumerate $p : X \to \mathbb{P}$ and g enumerate $q : X \to \mathbb{P}$.

1. $\lambda\langle n, m\rangle$. **if** $(f\,n, g\,m)$ **is** $(\mathsf{Some}\,x, \mathsf{Some}\,y)$ **then if** dxy **then** $\mathsf{Some}\,x$ **else** None **else** None enumerates $\lambda x.\ px \wedge qx$
2. $\lambda\langle n, m\rangle$. **if** n **is** 0 **then** $f\,m$ **else** $g\,m$ enumerates $\lambda x.\ px \vee qx$.

$\langle n, m\rangle$ → Sec. A.1, Page 207

Corollary 4.16. Enumerable predicates are closed under pointwise conjunction and disjunction.

Lemma 4.17. Let f enumerate p and g enumerate q.

$p \times q$
→ Sec. A.4, Page 209

1. $p \times q$ is enumerated by

$$\lambda\langle n, m\rangle.\ \textbf{if } (f\,n, g\,m) \textbf{ is } (\mathsf{Some}\,x, \mathsf{Some}\,y) \textbf{ then } \mathsf{Some}\,(x, y) \textbf{ else } \mathsf{None}$$

2. $p + q$ is enumerated by

$$\lambda\langle n, m\rangle.\ \textbf{if } n \textbf{ is } 0 \textbf{ then } \mathsf{omap}\ \mathsf{inl}\ (f\,m) \textbf{ else } \mathsf{omap}\ \mathsf{inr}\ (g\,m)$$

where $\mathsf{omap}\,h\,z := \textbf{if } z \textbf{ is } \mathsf{Some}\,z \textbf{ then } \mathsf{Some}\,(h z) \textbf{ else } \mathsf{None}$.

Corollary 4.18. If p and q are enumerable, $p \times q$ and $p + q$ are enumerable.

Lemma 4.19. If f enumerates $p : X \to Y \to \mathbb{P}$ then one can construct enumerators for both projections, i.e. for $\lambda x.\exists y.\ p\,x\,y$ and $\lambda y.\exists x.\ p\,x\,y$.

Proof. Let $f : \mathbb{N} \to \mathbb{O}(X \times Y)$ enumerate p. Then

$$\lambda n.\textbf{if } f\,n \textbf{ is } \mathsf{Some}\,(x, y) \textbf{ then } \mathsf{Some}\,x \textbf{ else } \mathsf{None}$$

enumerates $\lambda x.\exists y.p\,x\,y$ and similarly for $\lambda y.\exists x.p\,x\,y$. ∎

Corollary 4.20. If p is enumerable, then $\lambda x.\exists y.p\,x\,y$ and $\lambda y.\exists x.p\,x\,y$ are enumerable.

inhabited
→ Sec. A.4, Page 209

Note that for inhabited predicates (and consequently for inhabited types) we do not need the $\mathbb{O}$ type in enumerators. A function $f : \mathbb{N} \to X$ is a **strong enumerator** for a predicate $p : X \to \mathbb{P}$ if $\forall x.\ p\,x \leftrightarrow \exists n. f\,n = x$.

Def. strong enumerator

A predicate is **strongly enumerable** if there exists a strong enumerator:

Def. strongly enumerable predicate

$$\mathcal{E}_+(p : X \to \mathbb{P}) := \exists f : \mathbb{N} \to X.\forall x.\ p\,x \leftrightarrow \exists n. f\,n = x$$

Fact 4.21. Let $p : X \to \mathbb{P}$.

1. If f enumerates p and $x_0 : X$ such that $p\,x_0$, then $\lambda n.\ \textbf{if } f\,n \textbf{ is } \mathsf{Some}\,x \textbf{ then } x \textbf{ else } x_0$ strongly enumerates p.
2. If f strongly enumerates p, then $\lambda n.\ \mathsf{Some}\,(f\,n)$ enumerates p.

Fact 4.22. $\mathcal{E}_+ p \leftrightarrow \mathcal{E} p \wedge \exists x.p\,x$

Def. parametrically enumerable predicate

A predicate $p : I \to X \to \mathbb{P}$ is **parametrically enumerable** if there exists a function $f : I \to \mathbb{N} \to \mathbb{O}X$ such that f_i enumerates p_i:

$$\mathcal{E}_2\ p := \exists f : I \to \mathbb{N} \to \mathbb{O}X.\forall i x.\ p_i x \leftrightarrow \exists n.\ f_i n = \mathsf{Some}\,x$$

Fact 4.23. Let $p : I \to X \to \mathbb{P}$.

1. If f enumerates p and g decides equality on I, then p is parametrically enumerated by $\lambda i n.\ \textbf{if } f\,n \textbf{ is } \mathsf{Some}\,(j, x) \textbf{ then if } g\,i\,j \textbf{ then } \mathsf{Some}\,x \textbf{ else } \mathsf{None} \textbf{ else } \mathsf{None}$.

2. If f parametrically enumerates p and g enumerates I, then p is enumerated by $\lambda\langle n,m\rangle.$ **if** gm **is** Some i **then if** $f\,i\,m$ **is** Some x **then** Some (i,x) **else** None **else** None.

Corollary 4.24. If X is enumerable and discrete and $p\colon I \to X \to \mathbb{P}$, then $\mathcal{E}_2\, p \leftrightarrow \mathcal{E}p$.

We will need the result that the graph of a total function over enumerable domain is enumerable:

Fact 4.25. Let X be enumerable and $f\colon X \to Y$. Then $\mathcal{E}(\lambda xy.\, f\,x = y)$.

Similar to Lemma 4.6 we can prove a choice principle for relations which have an enumerator and where the domain has an equality decider:

Lemma 4.26. Let $R\colon X \to Y \to \mathbb{P}$, d decide equality on X, e enumerate R, and R be total, i.e. $\forall x.\exists y.Rxy$. Then one can construct $f\colon X \to Y$ such that $\forall x.\, Rx(f\,x)$.[1]

Proof. We define $R'(x\colon X)(n\colon \mathbb{N}) := \exists y.\ en = \text{Some}(x,y)$. R' is total since R is total and e enumerates R. R' is decided by $\lambda xn.$ **if** en **is** Some (x',y) **then** $d(x',x)$ **else** false since d decides equality on X.

We use Lemma 4.6 for R' to obtain a function $g\colon X \to \mathbb{N}$ s.t. $\forall x.\ R'x(f\,x)$. Now $f\,x :=$ **if** $e(g\,x)$ **is** Some (x',y) **then** y **else** _, where _ is irrelevant because this case can never occur, fulfils $\forall x.\, Rx(f\,x)$. ∎

4.2.1 Enumerable types

Fact 4.27. $f\colon \mathbb{N} \to \mathbb{O}X$ enumerates X if and only if it enumerates $\lambda x\colon X.\top$.

Corollary 4.28. X is enumerable if and only if $\lambda x\colon X.\top$ is enumerable.

Fact 4.29. $\mathbb{N}$ is enumerated by $\lambda n.$ Some n and $\mathbb{B}$ is enumerated by $\lambda n.$ **if** n **is** 0 **then** Some true **else** Some false.

Corollary 4.30. $\mathbb{N}$ and $\mathbb{B}$ are enumerable.

Fact 4.31. Let f enumerate X, g enumerate Y, and let $Z\colon X \to \mathbb{T}$ and $h\colon \forall x\colon X.\mathbb{N} \to \mathbb{O}(Zx)$ such that hx enumerates Zx.

1. $\lambda\langle n,m\rangle.$ **if** $(f\,n, g\,m)$ **is** (Some x, Some y) **then** Some (x,y) **else** None enumerates $X \times Y$.
2. $\lambda n.$ **if** n **is** S n **then** $f\,n$ **else** None enumerates $\mathbb{O}X$.
3. $\lambda\langle n,m\rangle.$ **if** $f\,n$ **is** Some x **then if** hxm **is** Some y **then** Some (x,y) **else** None **else** None enumerates $\Sigma x\colon X.\, Zx$.
4. $\lambda\langle n,m\rangle.$ **if** n **is** 0 **then** lift $f\ m$ inl **else** lift $g\ m$ inr enumerates $X + Y$, where lift $h\,n$ ins := **if** hn **is** Some z **then** Some (ins z) **else** None.

Corollary 4.32. Enumerable types are closed under pairs, sums, and options.

We postpone the closure of enumerable types under lists to the next section.

We will frequently require types to be discrete and enumerable in the following chapters. Such types are called **datatypes** in [73]. Equivalently to imposing that a type is discrete and enumerable, one can ask for a retraction into natural numbers:

[1] A similar result was anticipated by Larchey-Wendling [150], who formulated it for μ-recursively enumerable instead of synthetically enumerable predicates.

retraction
→ Sec. A.4, Page 209

Lemma 4.33. The following hold:

1. If d decides equality on X and f enumerates X, then

$$(\lambda x.\ \pi_1(\mu_{\mathbb{N}}(\lambda n.\ \textbf{if } f\,n \textbf{ is } \mathsf{Some}\, y \textbf{ then } d\,x\,y \textbf{ else } \mathsf{false})), f)$$

   is a retraction from X to $\mathbb{N}$.
2. If (I,R) is a retraction from X to $\mathbb{N}$, then $\lambda x_1 x_2.Ix_1 =_{\mathbb{B}} Ix_2$ decides equality on X.
3. If (I,R) is a retraction from X to $\mathbb{N}$, then R enumerates X.

Corollary 4.34. X is discrete and enumerable if and only if there exists a retraction from X to $\mathbb{N}$.

4.3 List enumerability

Some closure properties of enumerable predicates and types, e.g. closure of enumerable types under lists are hard to show directly. We thus introduce list enumerability as a tool. List enumerability is equivalent to enumerability, but often easier to treat in proofs, and we will also use it later to show the enumerability of concrete predicates. A function $L:\mathbb{N}\to\mathbb{L}X$ is a **list enumerator** for $p:X\to\mathbb{P}$ if $\forall x.\ px \leftrightarrow \exists n.\ x\in Ln$.

Def. list enumerator

Def. list enumerable

A predicate p is **list enumerable** if there exists a list enumerator:

$$\mathcal{E}_{\mathbb{L}}p := \exists L:\mathbb{N}\to\mathbb{L}X.\forall x.\ px \leftrightarrow \exists n.x\in Ln$$

Similarly, a function $L:\mathbb{N}\to\mathbb{L}X$ is a list enumerator for a type X if $\forall x.\exists n.x\in Ln$.

Fact 4.35. If f is an enumerator for p then $\lambda n.$ **if** $f\,n$ **is** $\mathsf{Some}\,x$ **then** $[x]$ **else** $[\,]$ is a list enumerator for p.

_[_] → Sec. A.1, Page 207

Fact 4.36. If L is a list enumerator for p then $\lambda\langle n,m\rangle.\ (en)[m]$ is an enumerator for p.

Corollary 4.37. p is enumerable if and only if it is list enumerable: $\mathcal{E}p \leftrightarrow \mathcal{E}_{\mathbb{L}}p$.

Def. cumulative

Closure constructions for both types and predicates are easier when working with cumulative list enumerators. A function $L:\mathbb{N}\to\mathbb{L}X$ is **cumulative** if $\forall n.\exists l.\ L(\mathsf{S}n) = Ln +\!\!+ l$.

Fact 4.38. If L is cumulative and $m\geq n$ then $\exists l.\ Lm = Ln +\!\!+ l$.

We define an operation $\mathsf{cumul}:(\mathbb{N}\to\mathbb{L}X)\to(\mathbb{N}\to\mathbb{L}X)$ as

$$\mathsf{cumul}\,L\ 0 := L0$$
$$\mathsf{cumul}\,L\ (\mathsf{S}n) := \mathsf{cumul}\,L\ n +\!\!+ L(\mathsf{S}n)$$

Fact 4.39. $\mathsf{cumul}\,L$ is cumulative and we have $(\exists n.x\in Ln)\leftrightarrow(\exists n.x\in\mathsf{cumul}\,Ln)$.

Lemma 4.40. If f is a list enumerator for X one can construct a list enumerator for $\mathbb{L}X$.

Proof. Let L_X be a list enumerator for X. Define

$$L0 := [\,]$$

$$L(\mathsf{S}n) := Ln +\!\!+ [x :: l \mid (x, L) \in \mathsf{cumul}\, L_X n \times Ln]$$

Clearly L is cumulative. $\forall l{:}\mathbb{L}X.\exists n.l \in Ln$ follows by induction on l. ∎

Corollary 4.41. List enumerable types and enumerable types are closed under lists.

4.4 Semi-decidability

Semi-decidable predicates are traditionally defined as the domain of a partial function from natural numbers to natural numbers. Since only the domain matters and not the results of the function, we do not use partial functions and instead directly define $f : X \to \mathbb{N} \to \mathbb{B}$ to be a **semi-decider** for a predicate $p : X \to \mathbb{P}$ if $\forall x.\ px \leftrightarrow \exists n.\ f\,x\,n = \mathsf{true}$. We introduce the notion of a co-semi-decider, where $f : X \to \mathbb{N} \to \mathbb{B}$ is a **co-semi-decider** for a predicate $p : X \to \mathbb{P}$ if $\forall x.\ px \leftrightarrow \forall n.\ f\,x\,n = \mathsf{false}$.

Def. semi-decider

Def. co-semi-decider

A predicate is **semi-decidable** if there exists a semi-decider, and **co-semi-decidable** if there exists a co-semi-decider:

Def. semi-decidable predciate

Def. co-semi-decidable predciate

$$\mathcal{S}(p{:}X \to \mathbb{P}) := \exists f{:}X \to \mathbb{N} \to \mathbb{B}.\forall x.\ px \leftrightarrow \exists n.\ f\,x\,n = \mathsf{true}$$

$$\overline{\mathcal{S}}p := \exists f{:}X \to \mathbb{N} \to \mathbb{B}.\forall x.\ px \leftrightarrow \forall n.f\,x\,n = \mathsf{false}$$

In the literature propositions P such that $P \leftrightarrow (\exists n.f\,n = \mathsf{true})$ are often called Σ^0_1 or "simply existential", and P such that $P \leftrightarrow (\forall n.f\,n = \mathsf{false})$ are called Π^0_1 or "simply universal". In our setting, semi-decidable predicates are pointwise Σ^0_1, and co-semi-decidable predicates are pointwise Π^0_1.

As for decidability we lift semi-decidability to n-ary predicates, i.e. semi-deciders of predicates $p{:}X_1 \to \cdots \to X_n$ are functions $f : X_1 \to \cdots \to X_n \to \mathbb{N} \to \mathbb{P}$.

Fact 4.42. Let $\forall x.(\exists n.\ f\,x\,n = \mathsf{true}) \leftrightarrow (\exists n.\ g\,x\,n = \mathsf{true})$, $p \equiv_{X\to\mathbb{P}} q$, and $p', q' : X \to \mathbb{P}$.

≡ → Sec. A.3, Page 208

1. f semi-decides p if and only if g semi-decides q.
2. p is semi-decidable if and only if q is semi-decidable.
3. If f semi-decides p' and g semi-decides q', then $p' \equiv_{X\to\mathbb{P}} q'$.

The complement of semi-decidable predicates is co-semi-decidable, due to the following intuitionistic tautology:

Fact 4.43. $\forall n.\ f\,n = \mathsf{false} \leftrightarrow \neg\exists n.\ f\,n = \mathsf{true}$

Corollary 4.44. Co-semi-decidable predicates are stable.

Fact 4.45. If f is a semi-decider for p, then f is a co-semi-decider for $\overline{p}$.

Corollary 4.46. The complement of semi-decidable predicates is co-semi-decidable: $\mathcal{S}p \to \overline{\mathcal{S}}\,\overline{p}$.

Note that $\overline{\mathcal{S}}\overline{p} \to \mathcal{S}p$ seems to be not provable.

As for enumerable predicates, decidable predicates are semi-decidable and co-semi-decidable:

Fact 4.47. If f is a decider for p, $\lambda xn.\ f\,x$ is a semi-decider for p and $\lambda xn.\ \neg_{\mathbb{B}} f\,x$ is a co-semi-decider for p.

Corollary 4.48. Decidable predicates are semi-decidable and co-semi-decidable. Furthermore, the complement of decidable predicates is semi-decidable.

As mentioned, semi-decidable predicates are often called Σ_1^0 in the literature. This is due to the following observation, stating that semi-decidable predicates can be expressed as one existential quantification over a decidable predicate:

Fact 4.49. A predicate p is semidecidable if and only if there exists $q\colon \mathbb{N} \to X \to \mathbb{P}$ such that q is decidable and $\forall x.px \leftrightarrow \exists n.\, qnx$.

We now prove closure properties of semi-decidable and co-semi-decidable predicates:

Lemma 4.50. Let f and g be semi-deciders for p and q. Then one can construct a semi-decider for $\lambda x.\, px \wedge qx$.

Proof. One can construct a function $\bigvee_{\mathbb{B}}^n : (\mathbb{N} \to \mathbb{B}) \to \mathbb{B}$ s.t. $\bigvee_{\mathbb{B}}^n h = \mathsf{true} \leftrightarrow \exists i \leq n.\, hi = \mathsf{true}$. Then $\lambda x.\, (\bigvee_{\mathbb{B}}^n (f\,x)) \wedge_{\mathbb{B}} (\bigvee_{\mathbb{B}}^n (g\,x))$ is a semi-decider for $\lambda x.\, px \wedge qx$ because $px \wedge qx \leftrightarrow \exists n.(\exists i \leq n.\, f\,xi = \mathsf{true}) \wedge (\exists i \leq n.\, g\,xi = \mathsf{true})$. ∎

Lemma 4.51. Let f and g be semi-deciders for p and q. Then $\lambda xn.f\,xn \vee_{\mathbb{B}} g\,xn$ is a semi-decider for $\lambda x.\, px \vee qx$.

Fact 4.52. Let f be a co-semi-decider for p and g be a co-semi-decider for q. Then $\lambda xn.\, f\,xn \vee_{\mathbb{B}} g\,xn$ is a co-semi-decider for $\lambda x.\, px \wedge qx$.

Corollary 4.53. The following hold:

1. Semi-decidable predicates are closed under pointwise conjunction and disjunction.
2. Co-semi-decidable predicates are closed under pointwise conjunction.

On predicates over natural numbers, enumerability and semi-decidability are equivalent. However, when generalising the notions to arbitrary base types instead of natural numbers, as we do here, one has to impose conditions on the base types to prove the equivalence. In particular, the two directions of the equivalence can be proved for strictly more types than natural numbers:

Fact 4.54. Let f be a semi-decider for $p\colon X \to \mathbb{T}$ and e enumerate X. Then

$$\lambda\langle n, m\rangle.\textbf{if } en \textbf{ is } \mathsf{Some}\, x \textbf{ then if } f\,xm \textbf{ then } \mathsf{Some}\, x \textbf{ else } \mathsf{None} \textbf{ else } \mathsf{None}$$

enumerates p.

Corollary 4.55. If p is semi-decidable and X is enumerable then p is enumerable.

Fact 4.56. Let e enumerate p and d decide equality on X. Then

$$\lambda xn.\textbf{if } en \textbf{ is } \mathsf{Some}\, y \textbf{ then } dxy \textbf{ else } \mathsf{false}$$

is a semi-decider for p.

Corollary 4.57. If p is enumerable and X is discrete then p is semi-decidable.

As before, we can construct choice functions for semi-decidable relations. In fact, this direct strenghtening of Lemma 4.6.[2]

Lemma 4.58. Let $R : X \to \mathbb{N} \to \mathbb{P}$ be total, i.e. $\forall x.\exists n.\ Rxn$, and let g be a semi-decider for R. Then one can construct a function $f : X \to \mathbb{N}$ such that $\forall x.\ Rx(fx)$.

Proof. We define $R'x\langle n, m\rangle := g(x, n)m = \mathsf{true}$. R' is total, since R is total and g semi-decides R. R' is decided by $\lambda x\langle n, m\rangle.\ g(x, n)m$.

We use Lemma 4.6 for R' to obtain a choice function $f' : X \to \mathbb{N}$ s.t. $\forall x.\ R'x(f'x)$. The wanted choice function for R then is $\lambda x.\ (\lambda\langle n, m\rangle.\ n)(f'x)$. ∎

For co-semi-decidable relations no choice principle seems to be provable, we will discuss this in more detail in Chapter 7.

4.5 Partial functions

In textbooks, recursive enumerability is often defined in terms of partial computable functions. We could not do so, because in type theory all definable functions are total by definition. Modelling partiality by functional relations $R : A \to B \to \mathbb{P}$ is not an option for us, because they subsume non-computable functions.

An alternative is to resort to step-indexed functions $A \to (\mathbb{N} \to \mathbb{O}B)$, sometimes with the additional constraint that once a value is found, i.e. $fan = \mathsf{Some}\, v$, it is not changed by increasing the step index. This approach was for instance pioneered by Richman [198] in constructive logic. Escardó and Knapp [65] provide a comprehensive overview of other approaches.

To show that indeed enumerability and semi-decidability can be defined using various models of partial functions in type theory, we assume a type $\mathsf{part}\, A$ for $A : \mathbb{T}$ and write $A \rightharpoonup B$ for $A \to \mathsf{part}\, B$. We for now abstract away from a concrete implementation of partial functions to obtain the result for all possible implementations. At the end of this section, we then discuss one implementation of partial functions in type theory, namely as stationary sequences $\mathbb{N} \to \mathbb{O}A$.

Def. $A \rightharpoonup B$

Before that, we specify several required operations on the type $\mathsf{part}\, A$, and then show that

1. Enumerable predicates can be defined as the co-domain of partial functions from the natural numbers:
$$\mathcal{E}p \leftrightarrow \exists f : \mathbb{N} \rightharpoonup X.\forall x : X.\ px \leftrightarrow \exists n.\ fn \triangleright x$$
2. Semi-decidable predicates can be defined as the domain of arbitrary partial functions:
$$\mathcal{S}p \leftrightarrow \exists(Y : \mathbb{T})(f : X \rightharpoonup Y).\forall x.\ px \leftrightarrow \exists y.\ fx \triangleright y$$

4.5.1 Partial values

An abstract definition of partial values is centered around a type former $\mathsf{part} : \mathbb{T} \to \mathbb{T}$ and a definedness relation $\triangleright : \mathsf{part}A \to A \to \mathbb{P}$. We immediately define a termination relation in terms of definedness as $x\downarrow := \exists a.\ x \triangleright a$. Two partial values are **equivalent** if they have the same value (or no value): $x \equiv_{\mathsf{part}\, A} y := (\forall a.\ x \triangleright a \leftrightarrow y \triangleright a)$.

Def. $x\downarrow$

Def. equivalent partial values

[2] A formulation of this construction for disjunctions (equivalently: $R : X \to \mathbb{B} \to \mathbb{P}$) is due to Andrej Dudenhefner and was received in private communication.

part A: $\mathbb{T}$ $\triangleright$: part$A \to A \to \mathbb{P}$	$x \triangleright a_1 \to x \triangleright a_2 \to a_1 = a_2$
ret : $A \to$ part$A$ $\gg\!=$: part$A \to (A \to$ part$B) \to$ part$B$ undef: part$A$	ret $a \triangleright a$ $x \gg\!= f \triangleright b \leftrightarrow (\exists a.\, x \triangleright a \wedge f\, a \triangleright b)$ $\nexists a.$undef $\triangleright a$
μ: $(\mathbb{N} \to \mathbb{B}) \to$ part$\mathbb{N}$	$\mu f \triangleright n \leftrightarrow f\, n =$ true $\wedge$ $\forall m < n.\, f\, m =$ false
seval: part$A \to \mathbb{N} \to \mathbb{O}A$	$x \triangleright a \leftrightarrow \exists n.$ seval $x n =$ Some a seval $x n =$ Some $a \to$ $m \geq n \to$ seval $x m =$ Some a

Figure 4.1.: Type and operations for partial values

We assume monadic structure for part (ret and $\gg\!=$), an undefined value (undef), a minimisation operation (μ), and a step-indexed evaluator (seval). The operations and their specifications are listed in Figure 4.1.

Lemma 4.59. There is eval: $\forall x$: part$A.x\downarrow \to A$ such that if $H: x \triangleright a$ we have eval $x\ H = a$.

Lemma 4.60. One can construct a parallelisation operator _ $\parallel$ _: part$A \to$ part$B \to$ part$(A + B)$ such that

1. $x \parallel y \triangleright$ inl $a \to x \triangleright a$
2. $x \parallel y \triangleright$ inr $b \to y \triangleright b$
3. $x\downarrow \vee y\downarrow \to (x \parallel y)\downarrow$

Similar to Fact 4.25, the graph of partial functions is enumerable:

Fact 4.61. Let X be an enumerable type. Then the graph of partial functions $f : X \rightharpoonup Y$ is enumerable: $\mathcal{E}(\lambda x y.f\, x \triangleright y)$.

If furthermore X and Y are discrete, the graph of partial functions is semi-decidable: $\mathcal{S}(\lambda x y.f\, x \triangleright y)$.

4.5.2 Enumerability and semi-decidability revisited

In textbooks, recursively enumerable predicates are defined as either the domain or co-domain of a partial function, and then the other alternative is shown equivalent. In our synthetic approach, the two alternatives are made explicit as semi-decidability (domain) and enumerability (co-domain), which we have however both modelled using total functions only.

However, one can define enumerability and semi-decidability based on partial functions also in the synthetic settings.

Recall that we defined enumerable predicates as the range of functions $f : \mathbb{N} \to \mathbb{O}X$ and proved that for non-empty predicates one can equivalently ask for a function $f : \mathbb{N} \to X$ in Fact 4.22. Alternatively, we could have asked for a function $f : \mathbb{N} \rightharpoonup X$:

Fact 4.62. $\mathcal{E}p \leftrightarrow \exists f : \mathbb{N} \rightharpoonup X. \forall x.\ px \leftrightarrow \exists n.\ f\,n \triangleright x$

Proof. If $f : \mathbb{N} \to \mathbb{O}X$ enumerates p, $\lambda n.$ **if** $f\,n$ **is** $\mathsf{Some}\,x$ **then** $\mathsf{ret}\,x$ **else** undef is a partial enumerator of type $\mathbb{N} \rightharpoonup X$.

If vice versa $f : \mathbb{N} \rightharpoonup X$ is a partial enumerator for p, $\lambda\langle n, m\rangle.\mathsf{seval}\,(f\,n)m$ enumerates p. ∎

And similarly, our definition of semi-decidable predicates in terms of total functions is equivalent to defining semi-decidable predicates as the domain of partial functions:

Fact 4.63. $\mathcal{S}p \leftrightarrow \exists Y (f : X \rightharpoonup Y). \forall x.\ px \leftrightarrow \exists y.\ f\,x \triangleright y$

Proof. If $f : X \to \mathbb{N} \to \mathbb{B}$ is a semi-decider for p, $\lambda x.\ \mu(\lambda n.\ \mathsf{ret}\,(f\,x\,n))$ has type $X \rightharpoonup \mathbb{N}$ and domain p.

Vice versa, if the domain of $f : X \rightharpoonup Y$ is p, $\lambda x n.$ **if** $\mathsf{seval}\,(f\,x)n$ **is** $\mathsf{Some}\,y$ **then** true **else** false semi-decides p. ∎

Lastly, we fix that a partial function $f : X \rightharpoonup Y$ computes a functional relation $R : X \rightsquigarrow Y$ if

$$\forall x y.\ Rxy \leftrightarrow f\,x \triangleright y.$$

4.5.3 Stationary sequences as partial values

We call $f : \mathbb{N} \to \mathbb{O}A$ a **stationary sequence** (or just **stationary**) if $\forall n a.\ f\,n = \mathsf{Some}\,a \to \forall m \geq n.\ f\,m = \mathsf{Some}\,a$. For instance, the always undefined function $\lambda n.\ \mathsf{None}$ is stationary.

Def. stationary sequence

We define the type of partial values as $\mathsf{part}\,A := \Sigma f : \mathbb{N} \to \mathbb{O}A.\ f$ *is stationary*. In this section, we use the letters f and g for stationary sequences. We define:

$$f \triangleright a := \exists n.\ \pi_1 f\,n = \mathsf{Some}\,a$$

This definition of definedness is deterministic, since f is stationary.

It is straightforward to define step-indexed evaluation as:

$$\mathsf{seval}\,f\,n := \pi_1 f\,n$$

Lemma 4.64. $f \triangleright a \leftrightarrow \exists n.\ \pi_1 f\,n = \mathsf{Some}\,a$

Proof. By computation, the two sides are equal. ∎

It is similarly straightforward to define the sequences for return, bind and the undefined value:

$$\mathsf{ret}\,a := (\lambda n.\ \mathsf{Some}\,a, H_{\mathsf{ret}})$$

$$\mathsf{undef} := (\lambda n.\ \mathsf{None}, H_{\mathsf{undef}})$$

$$f \gg\!= F := (\lambda n.\ \textbf{if}\ \pi_1 f\,n\ \textbf{is}\ \mathsf{Some}\,a\ \textbf{then}\ \pi_1(Fa)n\ \textbf{else}\ \mathsf{None}, H)$$

It is straightforward to provide proofs $H_{\mathsf{ret}}, H_{\mathsf{undef}}$, and H that all constructred sequences are indeed stationary. Correctness of ret and undef is easy, we only show the correctness proof for bind:

Lemma 4.65. $f \gg\!= F \triangleright b \leftrightarrow (\exists a.\, f \triangleright a \wedge Fa \triangleright b)$

Proof. The direction from left to right is straightforward.

For the direction from right to left, let $\pi_1 f n_1 = \mathsf{Some}\, a$ and $\pi_1(Fa)n_2 = \mathsf{Some}\, b$. Then $\pi_1(f \gg\!= F)(n_1 + n_2) = \mathsf{Some}\, b$. ∎

The most interesting construction is to implement μ.

We do so by implementing a function $\mathsf{mu}(F\colon \mathbb{N} \to \mathsf{part}\mathbb{B})(i\colon \mathbb{N})(u\colon \mathbb{N})$ where i is a step-index, and u is an upper bound. The function returns an element of the type

$$\mathsf{res} ::= \mathsf{Diverged} \mid \mathsf{Allfalse} \mid \mathsf{Trueat}\ (n\colon \mathbb{N})$$

such that the following correctness statement holds:

Fact 4.66. The following hold:

1. If $\mathsf{mu}\, F\, i\, u = \mathsf{Diverged}$ then $\exists m < u.\, \pi_1(Fm)i = \mathsf{None} \wedge \forall k < m.\, \pi_1(Fk)i \neq \mathsf{Some}\,\mathsf{true}$.
2. If $\mathsf{mu}\, F\, i\, u = \mathsf{Allfalse}$ then $\forall m < u.\pi_1(Fm)i = \mathsf{Some}\,\mathsf{false}$.
3. If $\mathsf{mu}\, F\, i\, u = \mathsf{Trueat}\ m$ then $m < u \wedge \pi_1(Fm)i = \mathsf{Some}\,\mathsf{true} \wedge \forall k < m.\, \pi_1(Fk)i \neq \mathsf{Some}\,\mathsf{false}$.

We can then define

$$\mu F := (\lambda n.\mathbf{if}\ \mathsf{mu}\, F\, n\, n\ \mathbf{is}\ \mathsf{Trueat}\ n\ \mathbf{then}\ \mathsf{Some}\, n\ \mathbf{else}\ \mathsf{None}, H)$$

where the proof H uses Fact 4.66.

Correctness of μ is then proved relying on the following fact:

Fact 4.67. Let $R\colon \mathbb{N} \to X \to \mathbb{P}$ such that $\forall m \leq n. \exists x.\, Rmx$. Then there exists $l\colon \mathbb{L}X$ such that R holds pointwisely on $[0, \ldots, n]$ and l.

Lemma 4.68. $\mu f \triangleright n \leftrightarrow f\, n = \mathsf{true} \wedge \forall m < n.\, f\, m = \mathsf{false}$

Proof. The forward direction is easy. The backward direction uses Facts 4.66 and 4.67 and essentially consists in proving that the conclusions of Fact 4.66 are mutually excluding. ∎

We can use μ to define a function mkstat to turn any function $f\colon \mathbb{N} \to \mathbb{O}A$ into a stationary sequence:

$$\mathsf{mkstat} f := \mu(\lambda n.\mathbf{if}\ f\, n\ \mathbf{is}\ \mathsf{Some}\, a\ \mathbf{then}\ \mathsf{true}\ \mathbf{else}\ \mathsf{false}) \gg\!= \lambda n.\mathbf{if}\ f\, n\ \mathbf{is}\ \mathsf{Some}\, a\ \mathbf{then}\ \mathsf{ret}\, a\ \mathbf{else}\ \mathsf{undef}$$

Fact 4.69. $\mathsf{mkstat} f \triangleright a \leftrightarrow \exists n.\, f\, n = \mathsf{Some}\, a \wedge \forall m < n.\, f\, n = \mathsf{None}$

4.6 On $\exists$ vs. Σ

We defined all synthetic computability notions using existential quantification $\exists$ in $\mathbb{P}$, and thus for instance enumerability is a proposition, i.e. $\mathcal{E}p\colon \mathbb{P}$. On a first view, an alternative would have been to define

$$\mathcal{E}_\Sigma p\colon \mathbb{T} := \Sigma f\colon \mathbb{N} \to \mathbb{O}X. \forall x\colon X.\, px \leftrightarrow \exists n. f\, n = \mathsf{Some}\, x$$

However, this is not a faithful synthetic rendering of the textbook notion of enumerability, because it would yield an unsound synthetic approach. In Chapter 2, we have discussed that in a

synthetic approach to mathematics, if the synthetic rendering of a logical formula P is provable in the synthetic setting, then P should be provable in textbook approaches to computability.

To demonstrate the issue, we look at the following statement, which is false in analytic textbook presentations of computability theory:

> *Let $A_{i\in\mathbb{N}}$ be a famility of sets of natural numbers such that every A_i is recursively enumerable. Then $\bigcup_{i\in\mathbb{N}} A_i$ is recursively enumerable.*

For the synthetic analogue we model families of sets of natural numbers $A_{i\in\mathbb{N}}$ as predicates $p\colon\mathbb{N}\to\mathbb{N}\to\mathbb{P}$, and a big union $\bigcup_{i\in\mathbb{N}} A_i$ as $\lambda x.\exists n.\ pnx$. If we now model "recursively enumerable" as $\mathcal{E}_\Sigma$, we can prove this statement:

Lemma 4.70. Let $p\colon\mathbb{N}\to X\to\mathbb{P}$ such that $\forall n.\mathcal{E}_\Sigma(pn)$. Then $\mathcal{E}_\Sigma(\lambda x.\exists n.\ pnx)$.

Proof. Let $F\colon\forall n.\mathcal{E}_\Sigma(pn)$. Then $\lambda\langle n,m\rangle.\pi_1(Fn)m$ enumerates $\lambda x.\exists n.pnx$. ∎

Note that the Lemma becomes provable because modelling recursive enumerability as $\mathcal{E}_\Sigma$ changes the meaning of $\forall i.\ A_i$*is recursively enumerable* to mean that A_i is parametrically enumerable, i.e. "there is a fixed computable function $f:\mathbb{N}\to\mathbb{N}\to\mathbb{O}X$ such that for all i, fi enumerates A_i", which is strictly stronger than the intended meaning "for all i there exists a computable function $f_i:\mathbb{N}\to\mathbb{O}X$ enumerating A_i".

However, as mentioned, the statement is clearly *false* in traditional computability theory. To see this, we define the family of sets

$$A_i := \begin{cases} \{i\} & \textit{if the i-th Turing machine does not halt on the empty tape} \\ \{\ \} & \textit{otherwise} \end{cases}$$

Now for every i, A_i is a sub-finite set and thus classically finite. Thus, every A_i is recursively enumerable (similar to Corollary 4.11). However, we have:

$$\bigcup_{i\in\mathbb{N}} A_i = \{i \mid \textit{the i-th Turing machine does not halt on the empty tape}\}$$

Note that the set on the right is the complement of the specialized halting problem – which is not recursively enumerable – and thus the statement above is indeed *false* in traditional computability theory.

Thus $\mathcal{E}_\Sigma$ is not a faithful synthetic definition of enumerability in type theory and we have to choose $\mathcal{E}p$ as definition of enumerability. To be able to develop basic computability theory like done in this chapter, we accordingly also have to place the definition of enumerability and all other notions in $\mathbb{P}$.

We conclude by remarking that Lemma 4.70 can be proved for the propositional definition of decidability under the axiom of countable choice. We will discuss the compatibility of axioms with synthetic computability in Chapter 7 in more detail.

CHAPTER 5

Reducibility

The concept of reducibility of one problem to another was prevalent already in early developments of computability. Intuitively, a problem p is reducible to a problem q if a decider for q can be computationally turned into a decider for p. Conversely, if p is reducible to q and p is undecidable, q is undecidable as well.

The first formal definition of reducibility is due to Turing [230] and based on oracles, the terminology "Turing reducibility" was first used in the seminal paper by Post [189]. Post sets out to analyse the order structure of degrees of enumerable sets, a degree being an equivalence class of problems under reducibility. His lead question is whether there is an enumerable but undecidable set with degree strictly below the degree of the halting problem, and since he is unable to do so, develops notions of reducibility for which he can answer the question.

[230] Turing. 1939. Systems of logic based on ordinals.

[189] Post. 1944. Recursively enumerable sets of positive integers and their decision problems.

In this chapter we introduce Post's notions of one-one, many-one, and truth-table reducibility, and synthetically prove positive results about them (i.e. inclusion or existence results, but no non-inclusion or non-existence results). In Chapter 8 we will also settle Post's problems w.r.t. the latter two notions, i.e. find enumerable but undecidable predicates to which the halting problem is not many-one and truth-table reducible. As corollaries, we first obtain that all of the mentioned reducibility notions differ, even on enumerable, undecidable predicates, and secondly that not all undecidability proofs work by reduction from the halting problem.

For the presentation in this chapter, we largely follow the textbooks by Rogers [202], Soare [210], and Cutland [44]. Besides establishing that one-one, many-one, and truth-table reducibility are indeed reducibilities (i.e. are pre-orders and transport decidability backwards) we also connect and characterise the notions in various ways: We show that many-one reducibility $p \preceq_m q$ can be equivalently characterised as one-one reducibility of $p \times X \preceq_1 p \times X$. Similarly, truth-table reducibility $p \preceq_{\mathrm{tt}} q$ can be expressed as $p^{\mathrm{tt}} \preceq_1 q^{\mathrm{tt}}$. Lastly, we give a fully synthetic proof of the Myhill isomorphism theorem [174] stating that one-one equivalent predicates are in fact isomorphic.

[202] Rogers. 1987. Theory of Recursive Functions and Effective Computability.

[210] Soare. 1999. Recursively enumerable sets and degrees: A study of computable functions and computably generated sets.

[44] Cutland. 1980. Computability.

[174] Myhill. 1957. Creative sets.

Note that in this chapter we do not give constructions in theorem statements, i.e. instead of stating "If f is a many-one reduction from p to q and g is a decider for q, then ... is a decider for p" we only state "If q is decidable and $p \preceq_m q$, p is decidable.".

Outline In Sections 5.1 and 5.2 we prove basic results on many-one and one-one reductions. Section 5.3 contains the Myhill isomorphism theorem, while in Section 5.4 we prove basic results on truth-table reductions.

Publications Section 5.1 is largely based on [73]. The other section contain adapted pieces of text from [72], which were written solely by the author of this thesis.

[73] Forster, Kirst, and Smolka. "On synthetic undecidability in Coq, with an application to the Entscheidungsproblem." *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*.

[72] Forster, Jahn, and Smolka. "A Constructive and Synthetic Theory of Reducibility: Myhill's Isomorphism Theorem and Post's Problem for Many-one and Truth-table Reducibility in Coq." *Pre-print*.

5.1 Many-one reducibility

Def. many-one reduction

A function $f : X \to Y$ is a **many-one reduction** from p to q if $\forall x.\, px \leftrightarrow q(f\,x)$. A predicate p is **many-one reducible** to a predicate q if there exists a many-one reduction.

Def. many-one reducibility: $p \preceq_m q$

$$p \preceq_m q := \exists f : X \to Y.\forall x.\, px \leftrightarrow q(f\,x)$$

The intuition here is that f translates instances of p to instances of q. We extend many-one reducibility to n- and m-ary predicates by (implicit) uncurrying of the target problem, i.e. for $p : X_1 \to \cdots \to X_n \to \mathbb{P}$ and $q : Y_1 \to \cdots \to Y_m \to \mathbb{P}$ a many-one reduction from p to q is a function $f : X_1 \to \cdots \to X_n \to Y_1 \times \ldots Y_m$.

Fact 5.1. Many-one reducibility is a pre-order, i.e. reflexive and transitive.

Fact 5.2. If $p \preceq_m q$ and q is decidable then p is decidable.

Lemma 5.3. If $p \preceq_m q$ and q is semi-decidable then p is semi-decidable.

Proof. Let f reduce p to q and g be a semi-decider for q. Then $\lambda x n.g(f\,x)n$ semi-decides p. ∎

Lemma 5.4. Let X be enumerable, Y discrete, and $p : X \to \mathbb{P}$, $q : Y \to \mathbb{P}$. If $p \preceq_m q$ and q is enumerable then p is enumerable.

Proof. By Lemmas 4.55, 5.3, and 4.57. ∎

Fact 5.5. If $p \preceq_m q$ then $\overline{p} \preceq_m \overline{q}$.

However, note that many other traditional theorems concerning the complements of predicates require classical reasoning:

Lemma 5.6. All of the following are equivalent to the law of excluded middle LEM: $p \preceq_m \overline{\overline{p}}$, $\overline{\overline{p}} \preceq_m p$, and $\overline{p} \preceq_m \overline{q} \to p \preceq_m q$.

Proof. Take $p(x : \mathbb{N}) := P$ and $q(x : \mathbb{N}) := \neg\neg P$. Then $\neg\neg P \to P$ in all cases. ∎

Many-one reducibility forms an upper semi-lattice:

Lemma 5.7. Let $p : X \to \mathbb{P}$ and $q : Y \to \mathbb{P}$. Then for $p + q : X + Y \to \mathbb{P}$ we have $p \preceq_m p + q$, $q \preceq_m p + q$, and for all r if $p \preceq_m r$ and $q \preceq_m r$ then $p + q \preceq_m r$.

→ Sec. A.4, Page 209

Proof. (1) and (2) are immediate. For (3), let $r : Z \to \mathbb{P}$, $f : X \to Z$ reduce p to r, and $g : Y \to Z$ reduce q to r. $h(\mathsf{inl}\,x) := f\,x$, $h(\mathsf{inr}\,y) := g\,y$ reduces $p + q$ to r. ∎

Recall that a predicate p is stable if $\forall x.\neg\neg px \to px$.

Fact 5.8. If q is stable and $p \preceq_m q$ then p is stable.

stable predicate

→ Sec. A.4, Page 209

In textbooks, the halting problem for the chosen model of computation plays a central role to explain the order structure of many-one reducibility. When excluding trivial predicates, we have that decidable predicates reduce to all semi-decidable predicates, and all semi-decidable predicates reduce to the halting problem. Synthetically, we can for now only partially replicate this result:

$$\mathcal{K}_{\mathbb{N}\to\mathbb{B}} := \lambda f : \mathbb{N} \to \mathbb{B}.\exists n.\, f\,n = \mathsf{true}$$

Fact 5.9. If p is decidable and q is nontrivial then $p \preceq_m q$.

Fact 5.10. $\mathcal{S}p \leftrightarrow p \preceq_m \mathcal{K}_{\mathbb{N}\to\mathbb{B}}$

In general, we call predicates q on enumerable, discrete types such that $\mathcal{E}p \to p \preceq_m q$ **many-one complete**. Note that $\mathcal{K}_{\mathbb{N}\to\mathbb{B}}$ is not many-one complete, since $\mathcal{K}_{\mathbb{N}\to\mathbb{B}} : (\mathbb{N}\to\mathbb{B})\to\mathbb{P}$, and $\mathbb{N}\to\mathbb{B}$ is not an enumerable type. We will construct and discuss a many-one complete predicate based on axioms in Chapter 6.

Def. many-one complete

If $p \preceq_m q$ and $q \preceq_m p$, we say that p and q are many-one equivalent or m**-equivalent**, and write $p \equiv_m q$.

Def. m-equivalent

5.2 One-one reducibility

A special case of many-one reducibilitity is one-one reducibility. An injection $f : X \to Y$ is a **one-one reduction** from p to q if f is a many-one reduction. A predicate $p : X \to \mathbb{P}$ is **one-one reducible** to a predicate $q : Y \to \mathbb{P}$ if there exists a many-one reduction.

Def. one-one reduction

injection
→ Sec. A.3, Page 209

$$p \preceq_1 q := \exists f : X \to Y.\ f \text{ is injective} \wedge \forall x.\ px \leftrightarrow q(f\,x)$$

Def. one-one reducibility: $p \preceq_1 q$

If $p \preceq_1 q$ and $q \preceq_1 p$, we say that p and q are one-one equivalent or 1**-equivalent**, and write $p \equiv_1 q$.

Def. 1-equivalent

Lemma 5.11. One-one reducibility is a pre-order, i.e. reflexive and transitive.

Proof. The identity and composition of injective functions are injective. ∎

Fact 5.12. If $p \preceq_1 q$ then $p \preceq_m q$.

Due to this fact, one-one reducibility also transports decidability, enumerability and semi-decidability. We only state the former:

Fact 5.13. If $p \preceq_1 q$ and q is decidable then p is decidable.

Lemma 5.14. $(\lambda x.x \neq 0) \preceq_m (\lambda x.x = 0)$ but $(\lambda x.x \neq 0) \not\preceq_1 (\lambda x.x = 0)$.

Proof. A many-one reduction is immediate by $\lambda x.$**if** x **is** 0 **then** 1 **else** 0.

However, given a one-one reduction f, we need to have $f\,1 = f\,2 = 0$. But since f is an injection we have $1 = 2$, contradiction. ∎

Note that the lemma leaves open whether there are two enumerable, but undecidable predicates p, q such that $p \preceq_m q$ but $p \not\preceq_1 q$, which is the more interesting case and will be settled in Chapter 8.

Many-one reducibility can be characterised in terms of one-one reducibility [202, §7.6 Theorem VIII]. The traditional proofs works via so called cylindrification and we generalise it slightly to apply to base types other than $\mathbb{N}$. In our setting, a cylindrification of a predicate $p : X \to \mathbb{P}$ is $p \times Z$ for an inhabited type Z, and predicates of the form $p \times Z$ in general are called cylinders. First, we can prove that a cylindrification of p is many-one equivalent to p:

Fact 5.15. Let $p : X \to \mathbb{P}$.

1. $p \preceq_1 p \times Z$ given $z : Z$

2. $p \times Z \preceq_m p$

Then, we can show that any many-one reduction to a cylinder can be turned into a one-one reduction:

Lemma 5.16. If $q \preceq_m p \times Z$ then $q \preceq_1 p \times Z$, provided an injection $g : Y \times X \to Z$.

Proof. Let f reduce q to $p \times Z$. Then $\lambda y.(\pi_1(f\,y), g(y, \pi_1(f\,y)))$ is the wanted one-one reduction. It is injective since g is injective and correct since f is correct. ∎

This suffices to establish the following characterisation:

Lemma 5.17. Let $p : X \to \mathbb{P}$, $q : Y \to \mathbb{P}$, $z : Z$, and $f : Y \times Z \hookrightarrow Z$. Then $q \preceq_m p \leftrightarrow q \times Z \preceq_1 p \times Z$.

Proof. The forward direction follows by Lemma 5.16 and Fact 5.15. The backwards direction is by Fact 5.15. ∎

Furthermore, the cylindrification of p is 1-maximal in the m-degree of p:

Lemma 5.18. Let $p : X \to \mathbb{P}$, $q : Y \to \mathbb{P}$, $z : Z$, $f : Y \to Z$, $q \equiv_m p$ and $p \preceq_1 q$. Then $q \preceq_1 p \times Z$.

Proof. Direct from Lemma 5.16 and Fact 5.15 (1). ∎

5.3 Myhill isomorphism theorem

[174] Myhill. 1957. Creative sets.

[195] Pradic and Brown. 2019. Cantor-Bernstein implies Excluded Middle.

The Myhill isomorphism theorem [174] is a generalisation of the restriction of the Cantor-Bernstein theorem to enumerable, discrete types. In general, the (inherently classical [195]) Cantor-Bernstein theorem constructs a bijection between sets A and B from two injections $A \to B$ and $B \to A$. When restricted to enumerable, discrete types, Cantor-Bernstein becomes fully constructive. As a generalisation, the Myhill isomorphism theorem states that 1-equivalent predicates $p : X \to \mathbb{P}$ and $q : Y \to \mathbb{P}$ (i.e. there are injective reductions between them, but the reductions are in no relation to each other) are isomorphic (i.e. there are reductions f, g between them such that $f(g\,y) = y$ and $g(f\,x) = x$).

The isomorphism theorem does not rely on universal machines and can thus be synthetically replicated without axioms.

We loosely follow Rogers [202, §7.4 Th. VI], where the isomorphism is constructed in stages. The stages are formed by **correspondence sequences** between predicates p and q, which are finitary bijections represented as lists $C : \mathbb{L}(X \times Y)$ such that

Def. correspondence sequences

(a) $\forall (x, y) \in C.\ px \leftrightarrow qy$

(b) $\forall (x, y_1) \in C. \forall (x, y_2) \in C.\ y_1 = y_2$

(c) $\forall (x_1, y) \in C. \forall (x_2, y) \in C.\ x_1 = x_2$

We write $x \in_1 C$ if x is an element of the first projection of C, and $y \in_2 C$ if y is an element of the second.

The crux of the theorem is that for any C as above such that $p \preceq_1 q$ and $x_0 \notin_1 C$ one can compute y_0 such that $(x_0, y_0) :: C$ is a correspondence sequence again, with no condition on p and q:

Lemma 5.19. Let f be a one-one reduction from p to q. One can construct a function $\mathsf{find} : \mathbb{L}(X \times Y) \to X \to Y$ such that if C is a correspondence sequence for p and q and $x_0 \notin_1 C$, then $\mathsf{find}\ C\ x_0 \notin_2 C$ and $px_0 \leftrightarrow q(\mathsf{find}\ C\ x_0)$.

Proof. We first define a function $\gamma : \mathbb{L}(X \times Y) \to X \to X$ recursive in $|C|$:

$$\gamma C x := x \qquad \text{if } f\,x \notin_2 C$$

$$\gamma C x := \gamma\,(\mathsf{filter}(\lambda t.\, t \neq_{\mathbb{B}} (x', f\,x))\,C)\,x' \qquad \text{if } (x', f\,x) \in C$$

For a correspondence sequence C between p and q and $x \notin_1 C$ we have (1) $px \leftrightarrow p(\gamma C x)$, (2) $\gamma C x = x$ or $\gamma X x \in_1 C$, and (3) $f(\gamma C x) \notin_2 C$. The proof is straightforward by induction on the length of C, exploiting the injectivity of f.

Now $\mathsf{find}\, C x_0 := f(\gamma C x_0)$ is the wanted function. ∎

[213] Sozeau and Mangin. 2019. Equations reloaded: High-level dependently-typed functional programming and proving in Coq.

The implementation of γ in Coq is greatly eased by using the Equations plugin [213].

For the rest of this section we fix enumerable, discrete types X and Y such that (I_X, R_X) and (I_Y, R_Y) are retractions from X and Y respectively to $\mathbb{N}$ by Corollary 4.34. We construct the isomorphism via a cumulative correspondence sequence C_n such that $I_X x < n \to x \in_1 C_n$ and $I_Y y < n \to y \in_2 C_n$.

$$C_0 := [\,]$$

$$C'_n := \begin{cases} (x, \mathsf{find}\, C_n\, x) :: C_n & \text{if } R_X n = \mathsf{Some}\, x \wedge x \notin_1 C_n \\ C_n & \text{otherwise} \end{cases}$$

$$C_{\mathsf{S}\,n} := \begin{cases} (\mathsf{find}\, \overleftrightarrow{C'_n}\, y, y) :: C'_n & \text{if } R_Y n = \mathsf{Some}\, y \wedge y \notin_2 C_n \\ C'_n & \text{otherwise} \end{cases}$$

where $\overleftrightarrow{C} := \mathsf{map}\,(\lambda(x,y).(y,x))\,C$ is a correspondence sequence for q and p if C is one for p and q.

Fact 5.20. C_n is a correspondence sequence for p and q such that

1. $n \leq m \to C_n \subseteq C_m$
2. $I_X x < n \to x \in_1 C_n$
3. $I_Y y < n \to y \in_2 C_n$

C_n now gives rise to the wanted isomorphism:

Lemma 5.21. There are functions $f' : X \to Y$ and $g' : Y \to X$ such that

1. $\forall x.\, px \leftrightarrow q(f'x)$ and $\forall y.\, qy \leftrightarrow p(g'y)$.
2. $g'(f'x) = x$ and $f'(g'y) = y$.

Proof. $f'x$ is defined as the unique y for which $(x,y) \in C_{\mathsf{S}\,(I_X x)}$ (which exists by Fact 5.20 (2) and is unique because $C_{\mathsf{S}\,(I_X x)}$ is a correpsondence sequence), and $g'y$ is symmetrically defined as the unique x for which $(x,y) \in C_{\mathsf{S}\,(I_Y y)}$.

(1) is immediate since C_n is a correspondence sequence. (2) is by case analysis whether $I_X x \leq I_Y y$ or vice versa. ∎

Theorem 5.22. Let X and Y be enumerable, discrete types, $p: X \to \mathbb{P}$, and $q: Y \to \mathbb{P}$. If $p \preceq_1 q$ and $q \preceq_1 p$, then there exist $f: X \to Y$ and $g: Y \to X$ such that for all $x: X$ and $y: Y$:

$$px \leftrightarrow q(fx), \quad qy \leftrightarrow p(gy), \quad g(fx) = x, \quad f(gy) = y$$

Corollary 5.23 (Computational Cantor-Bernstein). Let X and Y be enumerable, discrete types and $f: X \to Y$ and $g: Y \to X$ be injections. Then one can construct $f': X \to Y$ and $g': Y \to X$ such that for all $x: X$ and $y: Y$:

$$g'(f'x) = x, \qquad f'(g'y) = y$$

5.4 Truth-table reducibility

Recall that p is many-one reducible to q if a decision for px can be computed from one instance of q, namely $q(fx)$. Truth-table reducibility generalises this intuition: A predicate p is truth-table reducible to q if a decision for px can be computed by evaluating a boolean formula with atoms of the form qy_i for finitely many queries y_i. Equivalently, boolean formulas with n inputs can also be expressed as truth-tables with 2^n rows, explaining the name "truth-table reducibility". We model the type of truth-tables by defining $\mathsf{truthtable}: \mathbb{T} := \mathbb{LB}$. Given an assignment $l: \mathsf{truthtable}$ of length n, we use a canonical listing function $\mathsf{gen}: \mathbb{N} \to \mathbb{L}(\mathbb{LB})$ such that $|\mathsf{gen}\, n| = 2^n$ and $\forall l: \mathbb{LB}.\ l \in \mathsf{gen}\, n \leftrightarrow |l| = n$ to define

$$l \vDash T := \begin{cases} T[i] = \mathsf{Some}\,\mathsf{true} & \text{if } (\mathsf{gen}\,|l|)[i] = \mathsf{Some}\, l \\ \bot & \text{otherwise} \end{cases}$$

When convenient, we assume $l \vDash T: \mathbb{B}$ since $\lambda lT.\ l \vDash T$ is decidable. Any $f: \mathbb{LB} \to \mathbb{B}$ can be converted into the truth-table $\mathsf{map} f\,(\mathsf{gen}\, n)$ with n inputs such that $l \vDash \mathsf{map} f\,(\mathsf{gen}\, n) \leftrightarrow f\, l = \mathsf{true}$ provided $|l| = n$. On paper, we abuse notation and treat any function $f: \mathbb{LB} \to \mathbb{B}$ as truth-table.

Def. truth-table reduction

A function $f: X \to \mathbb{L}Y \times \mathsf{truthtable}$ is a **truth-table reduction** from p to q if $\forall x. px \leftrightarrow l \vDash \pi_2(fx)$ for all lists l which pointwisely reflect q on the query list $\pi_1(fx)$. A predicate $p: X \to \mathbb{P}$ is **truth-table reducible** to a predicate $q: Y \to \mathbb{P}$ if there is a truth-table reduction:

Def. truth-table reducibility: $p \preceq_{\mathrm{tt}} q$

Forall_2
→ Sec. A.1, Page 208

$$p \preceq_{\mathrm{tt}} q := \exists f: X \to \mathbb{L}Y \times \mathsf{truthtable}.$$
$$\forall xl.\ \mathsf{Forall}_2\,(\lambda yb.\ qy \leftrightarrow b = \mathsf{true})\,(\pi_1(fx))\,l$$
$$\to px \leftrightarrow l \vDash \pi_2(fx) \qquad (\text{"}p \text{ is tt-reducible to } q\text{"})$$

Proving that tt-reducibility is indeed a reducibility is straightforward:

Lemma 5.24. Truth-table reducibility is a pre-order, i.e. reflexive and transitive.

Proof. The function $\lambda x.([x], \lambda[b].b)$ reduces p to p.
The transitivity proof is slighly more intricate, we omit it here. ∎

Lemma 5.25. If $p \preceq_{\mathrm{tt}} q$ and q is decidable then p is decidable.

Proof. Let f reduce p to q and d be a decider for q. Then d can be used to supply the input for the truth-table for x, i.e. $\lambda x.\pi_2(fx)(\mathsf{map}\ d\ (\pi_1(fx)))$ is a decider for p. ∎

Lemma 5.26. If $p \preceq_m q$ then $p \preceq_{\mathrm{tt}} q$.

Proof. Let f be a many-one reduction from p to q. Then $\lambda x.([f\,x], \lambda\,[b].\ b)$ is a truth-table reduction from p to q. ∎

The converse direction is not true in general:

Lemma 5.27. $\lambda x.\top \preceq_{\mathrm{tt}} \lambda x.\bot$, but $\lambda x.\top \not\preceq_m \lambda x.\bot$.

Proof. $\lambda x.([\,], \lambda x.\ \mathsf{true})$ is a truth-table reduction.
Any many-one reduction yields $\top \leftrightarrow \bot$, which is contradictory. ∎

Corollary 5.28. Many-one and truth-table reducibility differ.

Later on we will also see enumerable but undecidable predicates where the reducibilities differ.

Lemma 5.29. $\overline{p} \preceq_{\mathrm{tt}} p$

Proof. By $\lambda x.([x], \lambda\,[b].\ \neg_{\mathbb{B}} b)$. ∎

Lemma 5.30. Truth-table reducibility is an upper semi-lattice.

Proof. The proof for many-one reducibility can be adapted easily. ∎

Lastly, we show how to characterise truth-table reducibility in terms of one-one reducibility. Given a predicate $p : X \to \mathbb{P}$ we define the predicate $p^{\mathrm{tt}} : \mathbb{L}X \times \mathsf{truthtable} \to \mathbb{P}$:

$$p^{\mathrm{tt}} := \lambda z.\ \forall l.\ \mathsf{Forall}_2\ (\lambda x\,b.\ px \leftrightarrow b = \mathsf{true})(\pi_1 z)\ l \to l \vdash \pi_2 z$$

The characterisation seems to be inherently non-constructive, we thus assume p to be stable.

Lemma 5.31. Let $p : X \to \mathbb{P}$, $q : Y \to \mathbb{P}$, $x_0 : X$, $y_0 : Y$, and $g : \mathbb{L}X \times \mathsf{truthtable} \to Y$ be an injection. If p is stable, then $p \preceq_{\mathrm{tt}} q \leftrightarrow p^{\mathrm{tt}} \preceq_1 q^{\mathrm{tt}}$.

Proof. We loosely follow the proof by Rogers [202, §8.4 Th. IX] and prove the following statements:

1. If p is stable then $p \preceq_1 p^{\mathrm{tt}}$.
2. $p^{\mathrm{tt}} \preceq_{\mathrm{tt}} p$.
3. Every reduction $p \preceq_{\mathrm{tt}} q$ can be given via an injective reduction function, provided an injection $f : X \to Y$ exists.
4. If p is stable, $f : X \to Y$ injective then $p \preceq_{\mathrm{tt}} q \to p \preceq_1 q^{\mathrm{tt}}$.

(1) and (2) are straightforward. For (3) assume $p \preceq_{\mathrm{tt}} q$ via a reduction function g and construct $g'x := (f\,x :: \pi_1(gx), \lambda\,b :: L.\ \pi_2(gx)L)$. g' is injective since f is injective. For (4), let $p \preceq_{\mathrm{tt}} q$ via an injection f by (3). Then f 1-reduces p to q^{tt}.
The claim from left to right follows using (1), (2), and (4), the direction from right to left needs (1) and (4). ∎

Corollary 5.32. If $p, q : \mathbb{N} \to \mathbb{P}$ and p stable, $p \preceq_{\mathrm{tt}} q \leftrightarrow p^{\mathrm{tt}} \preceq_1 q^{\mathrm{tt}}$.

CHAPTER 6

Axioms for synthetic computability

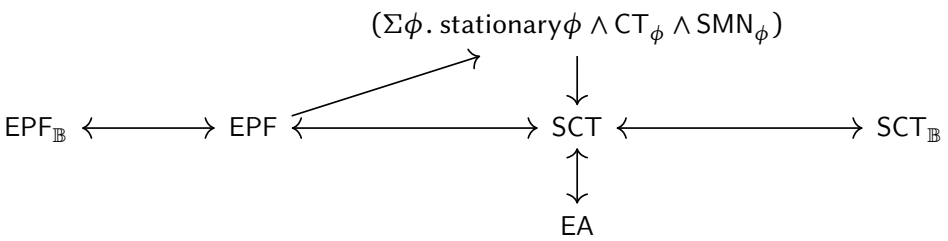
Results in textbooks on computability can roughly be split into positive results, like inclusions, reductions, or decidability results for certain problems, and into negative results, like undecidability, non-enumerability, or non-reducibility of problems. Until now we have proved various positive results by an intuitive identification of functions with computable functions, but we have neither proved negative results nor made this identification formally precise.

The development of negative results in textbooks starts with the definition of the self-halting problem, employing an encoding of programs as data. Via a direct diagonalisation, one shows that the complement of the self-halting problem is not enumerable. Afterwards, other undecidability or non-enumerability results can be deduced by positive intermediate lemmas: For instance the undecidability of the halting problem follows from the undecidability of the self-halting problem by many-one reduction.

While our introduced notions work well to verify these positive auxiliary results, we cannot prove negative results: Since CIC is consistent with strong classical axioms entailing the decidability of every problem, without axiomatic assumptions there is no hope in obtaining any undecidability result. To prove an initial negative result, we have to assume an axiom. We discuss various equivalent formulations of a parametric and synthetic version of the axiom CT ("Church's thesis"). The axiom CT_ϕ postulates that ϕ is a step-indexed interpreter universal for the function space $\mathbb{N}\to\mathbb{N}$, i.e. that every $f:\mathbb{N}\to\mathbb{N}$ has a code $c:\mathbb{N}$ such that ϕ_c agrees with f. We also consider an axiom SMN_ϕ, postulating an S_n^m operator for ϕ. We introduce five axioms equivalent to assuming ϕ such that $\mathsf{CT}_\phi \wedge \mathsf{SMN}_\phi$. Working with S_n^m operators in applications explicitly is tedious. We thus define all axioms via a respective notion of parametric universality. As a consequence, the statements of the axioms become more uniform and compact. At the same time, the axioms become easier to use in applications.

- SCT, postulating a step-indexed interpreter ϕ which is parametrically universal for $\mathbb{N}\to\mathbb{N}$. More precisely, every family of functions $f_i:\mathbb{N}\to\mathbb{N}$ parameterised by $i:\mathbb{N}$ has a coding function $\gamma:\mathbb{N}\to\mathbb{N}$ such that $\phi_{\gamma i}$ agrees with f_i for all parameters i.
- $\mathsf{SCT}_{\mathbb{B}}$, restricting the parametric universality of ϕ to functions $\mathbb{N}\to\mathbb{B}$.
- EPF, extending the parametric universality of ϕ to partial functions $\mathbb{N}\rightharpoonup\mathbb{N}$.
- $\mathsf{EPF}_{\mathbb{B}}$, restricting the parametric universality of ϕ to partial functions $\mathbb{N}\rightharpoonup\mathbb{B}$.
- EA, abstracting away from functions, focussing on parametrically enumerable predicates.

The following diagram depicts the connections we will prove:



All axioms have in common that they allow defining an enumerable but undecidable predicate $\mathcal{K}$. In Chapter 7 we discuss that SCT is a consequence of the constructivist axiom CT, and thus consistent in CIC but not in contradiction to the law of excluded middle. Thus, our formulations of SCT allow developing synthetic computability, agnostic towards classical logic.

As case studies we give two synthetic proofs of Rice's theorem [197]: One based on the axiom EPF, following the proof approach by Scott [204] relying on Rogers' fixed-point theorem [202], and one based on the axiom EA, establishing a many-one reduction from an undecidable problem. We furthermore provide solutions for Post's problem for many-one and truth-table reducibility in Chapter 8.

Outline We motivate and introduce CT_ϕ and SMN_ϕ in Section 6.1. SCT is introduced in Section 6.2, its variants EPF, $\mathsf{SCT}_{\mathbb{B}}$, and $\mathsf{EPF}_{\mathbb{B}}$ in Section 6.3. We use EPF to prove synthetic version of Rogers' and Rice's theorems in Section 6.5, and introduce EA in Section 6.4.

Publications Preliminary versions of the axioms discussed here can be found in the following publications.

[70] Forster. "Church's thesis and related axioms in Coq's type theory." *29th EACSL Annual Conference on Computer Science Logic (CSL 2021)*.

[72] Forster, Jahn, and Smolka. "A Constructive and Synthetic Theory of Reducibility: Myhill's Isomorphism Theorem and Post's Problem for Many-one and Truth-table Reducibility in Coq." *Pre-print*.

6.1 Church's thesis

[202] Rogers. 1987. Theory of Recursive Functions and Effective Computability.

Textbooks on computability start by defining a model of computation, Rogers [202] uses μ-recursive functions. As center of the theory, Rogers defines a step-indexed interpreter ϕ of all μ-recursively computable functions. An application $\phi_c^n x : \mathbb{ON}$ denotes executing the μ-recursive function with code c on input x for n steps.

Once some evidence is gathered, Rogers (as well as other authors) introduce the Church-Turing thesis, stating that all intuitively calculable functions are μ-recursively computable. Using the Church-Turing thesis, ϕ has the following (informal) universal property:

$$\forall f : \mathbb{N} \to \mathbb{N}.\ \textit{intuitively computable}\ f \to \exists c : \mathbb{N}. \forall x : \mathbb{N}. \exists n.\ \phi_c^n x = \mathsf{Some}\,(f\,x)$$

Note that the property is really only pseudo-formal: The notion of intuitive calculability is not made precise, which is exactly what allows ϕ to stay abstract for most of the development. Every invocation of the universality could be replaced by an individual construction of a (μ-recursive) program, but relying solely on the notion of intuitive calculability allows Rogers to build a theory based on a function ϕ which could equivalently be implemented in any other model of computation. Since not every function $f : \mathbb{N} \to \mathbb{N}$ in the classical set theory Rogers works in[1] is intuitively computable, every invocation of the universality of ϕ has to be checked individually to ensure that it is indeed for an intuitively calculable function.

We however do not work in classical set theory, but in CIC, a constructive system. As in all constructive systems, every definable function is intuitively calculable. It is thus natural to assume that the universal function ϕ is universal for *all* functions $f : \mathbb{N} \to \mathbb{N}$. For historical reasons, this axiom is called CT ("Church's thesis") [142, 228].

[142] Kreisel. 1965. Mathematical logic.

[228] Troelstra and van Dalen. 1988. Constructivism in mathematics. Vol. I.

[1] "We use the rules and conventions of classical two-valued logic (as is the common practice in other parts of mathematics), and we say that an object exists if its existence can be demonstrated within standard set theory. We include the axiom of choice as a principle of our set theory." Rogers [202, pg. 10, footnote †]

We define CT_ϕ parameterised in a step-indexed interpreter $\phi : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \to \mathbb{ON}$. As before, we write an evaluation of code c on input x for n steps as $\phi_c^n x$ instead of $\phi\, c\, x\, n$. For step-indexed interpreters, the sequence $\lambda n.\phi_c^n x$ is always stationary, i.e.

stationary sequence
→ Sec. 4.5, Page 41

$$\forall c x n_1 v.\ \phi_c^{n_1} x = \mathsf{Some}\, v \to \forall n_2.\ n_2 \geq n_1 \to \phi_c^{n_2} x = \mathsf{Some}\, v$$

Now CT_ϕ states that $\phi : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \to \mathbb{ON}$ is universal for *all* functions $f : \mathbb{N} \to \mathbb{N}$:

$$\mathsf{CT}_\phi := \forall f : \mathbb{N} \to \mathbb{N}. \exists c : \mathbb{N}. \forall x : \mathbb{N}. \exists n : \mathbb{N}.\ \phi_c^n x = \mathsf{Some}\,(f\, x)$$

Instead of seeing ϕ as a step-indexed interpreter, one can also see it as an enumeration of stationary functions from $\mathbb{N}$ to $\mathbb{N}$, which enumerates every total function f.

CT_ϕ is not provable in CIC, independent of the definition of ϕ. However, when ϕ is a step-indexed interpreter for a Turing-complete model of computation, one can give a meta-theoretical consistency proof of CT_ϕ, we discuss this in more detail in Section 7.1.

In contrast to textbook proofs, proofs of theorems based on CT_ϕ do not have to be individually checked for valid applications of the Church-Turing thesis.

As stated above, CT_ϕ applies to unary total functions, but is immediately extensible to n-ary functions $f : \mathbb{N}^n \to \mathbb{N}$ using pairing. Partial application for such n-ary functions is realised via the S_n^m theorem. We only state the case $m = n = 1$, which implies the general case:

$$\mathsf{SMN}_\phi := \Sigma\sigma : \mathbb{N} \to \mathbb{N} \to \mathbb{N}. \forall c x y v.\ (\exists n.\ \phi_{\sigma c x}^n y = \mathsf{Some}\, v) \leftrightarrow (\exists n.\ \phi_c^n \langle x, y\rangle = \mathsf{Some}\, v)$$

Note that we formulate SMN with a Σ rather than an $\exists$. For the results we prove in Chapter 8, the difference is largely cosmetic. For now, the formulation with Σ allows the construction of functions accessing σ directly, rather than only being able to prove the existence of functions based on σ.

The key property of CT_ϕ is that it allows the definition of an enumerable but undecidable problem:

Lemma 6.1. Let ϕ be stationary. Then $\mathsf{CT}_\phi \to \Sigma p : \mathbb{N} \to \mathbb{P}.\ \mathcal{S}p \wedge \neg\mathcal{S}\overline{p} \wedge \neg\mathcal{D}p \wedge \neg\mathcal{D}\overline{p}$.

Proof. One can define $pc := \exists nm.\ \phi_c^m \langle c, n\rangle = \mathsf{Some}\, 0$. p is semi-decided by $\lambda c\langle n, m\rangle$. **if** $\phi_c^m \langle c, n\rangle$ **is** $\mathsf{Some}\, 0$ **then** true **else** false. If $f : \mathbb{N} \to \mathbb{N} \to \mathbb{B}$ is a semi-decider for the complement of p, let c be its code w.r.t. ϕ. Then $pc \leftrightarrow \neg pc$, contradiction. Thus p is also not decidable. ∎

If one fixes ϕ to the step-indexed interpreter of μ-recursive functions ϕ_μ, then CT_{ϕ_μ} implies an identification of synthetic definitions of standard notions like decidability, enumerability, and reducibility with their respective analytic textbook definitions based on μ-recursive functions.

6.2 Synthetic Church's Thesis

By keeping ϕ abstract and assuming CT_ϕ, one never has to deal with encodings in a model of computation. However, formal proofs involving the SMN_ϕ axiom are tedious.

Def. synthetic Church's thesis

We identify the axiom **synthetic Church's thesis** SCT as a more convenient variant of $\mathsf{CT}_\phi \wedge \mathsf{SMN}_\phi$, which postulates a step-indexed interpreter ϕ **parametrically universal** for $\mathbb{N} \to \mathbb{N}$:

Def. parametrically universal

$$\mathsf{SCT} := \Sigma\phi\colon \mathbb{N}\to\mathbb{N}\to\mathbb{N}\to\mathbb{ON}.$$

$$(\forall cxn_1n_2v.\ \phi_c^{n_1}x = \mathsf{Some}\,v \to n_2 \geq n_1 \to \phi_c^{n_2}x = \mathsf{Some}\,v) \wedge$$

$$\forall(f\colon\mathbb{N}\to\mathbb{N}\to\mathbb{N}).\ \exists\gamma\colon\mathbb{N}\to\mathbb{N}.\ \forall ix.\exists n.\ \phi_{\gamma i}^n x = \mathsf{Some}\,(f_i x)$$

By parametrically universal we mean that for any family of functions $f_i\colon\mathbb{N}\to\mathbb{N}$ parameterised by $i\colon\mathbb{N}$, we obtain a coding function γ s.t. γi is the code of f_i, i.e. $\phi_{\gamma i}$ agrees with f_i.

The consistency of SCT follows from the consistency of CT formulated for a Turing-complete model of computation. For this purpose, we choose the weak call-by-value λ-calculus L, which we discuss in detail in Part IV. Conversely, one can recover non-parametric universality of ϕ from parametric universality:

Theorem 6.2. Let ϕ_{L} be a step-indexed interpreter for L. For any ϕ such that $\lambda n.\phi_c^n x$ is stationary we have the following:

1. $\mathsf{CT}_{\phi_{\mathsf{L}}} \to \Sigma\phi.\ \mathsf{CT}_\phi \wedge \mathsf{SMN}_\phi$
2. $\mathsf{CT}_\phi \to \mathsf{SMN}_\phi \to \mathsf{SCT}$
3. $\mathsf{SCT} \to \Sigma\phi.\ \mathsf{CT}_\phi$

Proof. (1) follows by proving $\mathsf{SMN}_{\phi_{\mathsf{L}}}$, which we do in Chapter 29, see Corollary 29.10.

For (2), let ϕ and σ be given. We prove that ϕ satisfies the condition in SCT. Let $f\colon\mathbb{N}\to\mathbb{N}\to\mathbb{N}$ be given. We obtain a code c for $\lambda\langle x,y\rangle.\ fxy$. Now define $\gamma x := \sigma cx$.

(3) is trivial by turning the unary function $f\colon\mathbb{N}\to\mathbb{N}$ into the (constant) family of functions $f'_x y := fy$. Now a coding function γ for f' allows to choose $\gamma 0$ as code for f. ∎

In Theorem 6.6 we prove that SCT also implies $\Sigma\phi.\ \mathsf{CT}_\phi \wedge \mathsf{SMN}_\phi$.

6.3 Variations of Synthetic Church's Thesis

We have defined SCT to postulate a step-indexed interpreter $\phi\colon\mathbb{N}\to(\mathbb{N}\to\mathbb{N}\to\mathbb{ON})$, parametrically universal for $\mathbb{N}\to\mathbb{N}$. In this section, we develop alternative equivalent definitions of SCT. In general, there are three obvious points where SCT can be modified.

1. The return type of ϕ can be stationary functions $\mathbb{N}\to(\mathbb{N}\to\mathbb{ON})$, partial functions $\mathbb{N}\rightharpoonup\mathbb{N}$, stationary functions $\mathbb{N}\to(\mathbb{N}\to\mathbb{OB})$, or partial functions $\mathbb{N}\rightharpoonup\mathbb{B}$,
2. ϕ can be postulated to be parametrically universal for $\mathbb{N}\to\mathbb{N}$, $\mathbb{N}\to\mathbb{B}$, $\mathbb{N}\rightharpoonup\mathbb{N}$, $\mathbb{N}\rightharpoonup\mathbb{B}$, or stationary functions $\mathbb{N}\to(\mathbb{N}\to\mathbb{ON})$ or $\mathbb{N}\to(\mathbb{N}\to\mathbb{OB})$.
3. Coding functions γ can be existentially quantified ($\exists$), computably obtained (Σ), or classically existentially quantified $\neg\neg\exists$.

For SCT, the return type of ϕ is stationary functions $\mathbb{N}\to(\mathbb{N}\to\mathbb{ON})$, ϕ is parametrically universal for $\mathbb{N}\to\mathbb{N}$, and γ is existentially quantified.

[12] Bauer. 2015. An injection from the Baire space to natural numbers.

For (1), it is important to see that letting ϕ return total functions is no option, since such an enumeration is inconsistent in any logic with functions,[2] even up to extensionality:

[2] Note that conversely an injection of $(\mathbb{N}\to\mathbb{N})\to\mathbb{N}$ can likely be consistently assumed [12].

Fact 6.3 (Cantor's theorem). There is no $e\colon \mathbb{N}\to(\mathbb{N}\to A)$ such that $\forall f\colon \mathbb{N}\to A.\ \exists c.ec \equiv_{\mathbb{N}\to A} f$ for $A=\mathbb{N}$ or $A=\mathbb{B}$.

For (3), the variant with Σ is consistent, but negates functional extensionality, we discuss this in Section 7.3. Variants with $\neg\neg\exists$ are often called *Weak* CT [169], we refrain from discussing it in this thesis.

[169] McCarty. 1991. Incompleteness in intuitionistic metamathematics..

In this section, we discuss how all other variations of SCT are equivalent, and single out three of them:

1. EPF, the enumerability of partial functions axiom, postulating $\theta\colon\mathbb{N}\to(\mathbb{N}\rightharpoonup\mathbb{N})$ parametrically universal for $\mathbb{N}\rightharpoonup\mathbb{N}$,
2. $\mathsf{SCT}_{\mathbb{B}}$, postulating $\phi\colon\mathbb{N}\to(\mathbb{N}\to(\mathbb{N}\to\mathbb{OB}))$ universal for $\mathbb{N}\to\mathbb{B}$, and
3. $\mathsf{EPF}_{\mathbb{B}}$, postulating $\theta\colon\mathbb{N}\to(\mathbb{N}\rightharpoonup\mathbb{B})$ universal for $\mathbb{N}\rightharpoonup\mathbb{B}$.

The **enumerability of partial functions axiom** EPF is defined as:

Def. enumerability of partial functions axiom

$$\mathsf{EPF} := \Sigma\theta\colon\mathbb{N}\to(\mathbb{N}\rightharpoonup\mathbb{N}).\forall f\colon\mathbb{N}\to\mathbb{N}\rightharpoonup\mathbb{N}.\exists\gamma\colon\mathbb{N}\to\mathbb{N}.\forall i.\ \theta_{\gamma i}\equiv_{\mathbb{N}\rightharpoonup\mathbb{N}} f_i$$

Instead of seeing θ as enumeration, we can also see θ as surjection from $\mathbb{N}$ to $\mathbb{N}\rightharpoonup\mathbb{N}$ up to $\equiv_{\mathbb{N}\to\mathbb{N}}$. Proving that $\mathsf{SCT}\leftrightarrow\mathsf{EPF}$ amounts to showing that any implementation of partial functions is equivalent to the implementation based on stationary sequences we gave in Section 4.5, and that any stationary function can be encoded in a total function $\mathbb{N}\to\mathbb{N}$ via pairing.

Theorem 6.4. $\mathsf{SCT}\leftrightarrow\mathsf{EPF}$

Proof. The direction from left to right is by observing that there is a function $\mathsf{mktotal}\colon(\mathbb{N}\to\mathbb{N}\rightharpoonup\mathbb{N})\to\mathbb{N}\to\mathbb{N}\to\mathbb{N}$ s.t. $f_i x\triangleright v\leftrightarrow\exists n.\ \mathsf{mktotal}\,f\,i\,\langle x,n\rangle = \mathsf{S}\,v$ using seval. We then define

$$\begin{aligned}\theta cx :=& (\mu(\lambda n.\ \textbf{if } \phi_c^n x \textbf{ is } \mathsf{Some}(\mathsf{S}\,v) \textbf{ then } \mathsf{ret}\,\mathsf{true} \textbf{ else } \mathsf{ret}\,\mathsf{false}))\\ &\gg\!= \lambda n.\ \textbf{if } \phi_c^n x \textbf{ is } \mathsf{Some}(\mathsf{S}\,v) \textbf{ then } \mathsf{ret}\,v \textbf{ else } \mathsf{undef}\end{aligned}$$

The direction from right to left constructs $\phi_c^n x := \mathsf{seval}\,(\theta_c y)\,n$. Let $f\colon\mathbb{N}\to\mathbb{N}\to\mathbb{N}$. Define the partial function $f_i' x := \mathsf{ret}\,(f_i x)$. Now a coding function γ for f' by EPF is a coding function for f to establish SCT. ∎

Instead of stating EPF as enumeration of partial functions, we can equivalently state it w.r.t. parameterised functional relations:

Fact 6.5. EPF is equivalent to the following:

$$\Sigma\theta\colon\mathbb{N}\to(\mathbb{N}\rightharpoonup\mathbb{N}).\ \forall R\colon\mathbb{N}\to(\mathbb{N}\rightsquigarrow\mathbb{N}).\ (\exists f.\forall i.\ f_i \textit{ computes } R_i)\to\exists\gamma.\forall i.\ \theta_{\gamma i} \textit{ computes } R_i$$

Theorem 6.6. $\mathsf{EPF}\to\Sigma\phi.\ \mathsf{CT}_\phi\wedge\mathsf{SMN}_\phi$

Proof. Let θ be given as in EPF and define $\phi_c^n x := \mathsf{seval}\,(\theta_c y)\,n$, which allows proving CT_ϕ as in Theorem 6.4. Let furthermore $f_{\langle c,x\rangle} y := \theta c\langle x,y\rangle$ and γ be a coding function for f by EPF. Define $Scx := \gamma\langle c,x\rangle$. We have

$$\theta_{Scx} y \equiv \theta_{\gamma\langle c,x\rangle} y \equiv \theta c\langle x,y\rangle$$

∎

We introduce $\mathsf{SCT}_{\mathbb{B}}$, postulating a step-indexed interpreter parametrically universal for $\mathbb{N} \to \mathbb{B}$:

$$\mathsf{SCT}_{\mathbb{B}} := \Sigma \phi : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \to \mathbb{O}\mathbb{B}.$$
$$(\forall c x n_1 n_2 v.\ \phi_c^{n_1} x = \mathsf{Some}\, v \to n_2 \geq n_1 \to \phi_c^{n_2} x = \mathsf{Some}\, v) \wedge$$
$$\forall f : \mathbb{N} \to \mathbb{N} \to \mathbb{B}.\ \exists \gamma. \forall i x. \exists n.\ \phi_{\gamma i}^{n} x = \mathsf{Some}\,(f_i x)$$

$\mathsf{SCT}_{\mathbb{B}}$ is equivalent to SCT. One direction is immediate since $\mathbb{B}$ is a retract of $\mathbb{N}$ (i.e. can be injectively embedded). The other direction follows by mapping the infinite sequence $f0, f1, f2, \ldots$ to the sequence

$$\mathsf{false}^{f0}\,\mathsf{true}\,\mathsf{false}^{f1}\,\mathsf{true}\,\mathsf{false}^{f2}\,\mathsf{true}\ldots$$

Theorem 6.7. $\mathsf{SCT}_{\mathbb{B}} \leftrightarrow \mathsf{SCT}$

Proof. The direction from right to left is trivial. For the converse direction, define $I : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{B}$ and $R : (\mathbb{N} \rightharpoonup \mathbb{B}) \to \mathbb{N} \rightharpoonup \mathbb{N}$ as

$$I(f : \mathbb{N} \to \mathbb{N}) x := [\mathsf{false}^n \mathsf{true} \mid n \in [f0, \ldots, fx, 1 + fx]] \qquad Rf0 := \mu f$$
$$Rf(\mathsf{S}\, x) := \mu f \gg\!= \lambda n.\, R(\lambda m.\, f(m + 1 + n))x$$

We have $(\forall x.\ gx \triangleright If x) \to \forall x.\ Rgx \triangleright fx$.

Given ϕ parametrically universal for $\mathbb{N} \to \mathbb{B}$ as in $\mathsf{SCT}_{\mathbb{B}}$, define $\phi' c := R(\lambda x.\ \phi c x)$. Now given $f : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ we use $\mathsf{SCT}_{\mathbb{B}}$ for $f_i' x := I f_i x$ to obtain γ.

By the characteristic property of R, we can choose γ for f as well to prove SCT. ∎

Def. parametric enumerability of partial boolean functions

Lastly, we define the **parametric enumerability of partial boolean functions** axiom

$$\mathsf{EPF}_{\mathbb{B}} := \Sigma \theta : \mathbb{N} \to (\mathbb{N} \rightharpoonup \mathbb{B}). \forall (f : \mathbb{N} \to \mathbb{N} \rightharpoonup \mathbb{B}). \exists \gamma : \mathbb{N} \to \mathbb{N}. \forall i.\ \theta_{\gamma i} \equiv_{\mathbb{N} \to \mathbb{B}} f_i$$

$\equiv_{\mathbb{N} \to \mathbb{B}}$ → Sec. 4.5, Page 39

Recall that $\theta_{\gamma i} \equiv_{\mathbb{N} \to \mathbb{B}} f_i$ if and only if $\forall x v.\ \theta_{\gamma i} x \triangleright v \leftrightarrow f_i x \triangleright v$. Proving $\mathsf{EPF}_{\mathbb{B}}$ equivalent to SCT is easiest done by proving the following:

Theorem 6.8. $\mathsf{EPF}_{\mathbb{B}} \leftrightarrow \mathsf{SCT}_{\mathbb{B}}$

Proof. Exactly as in Theorem 6.4. ∎

Using $\mathsf{EPF}_{\mathbb{B}}$ it is easy to establish an enumerable, undecidable problem:

Fact 6.9. $\mathsf{EPF}_{\mathbb{B}} \to \Sigma p : \mathbb{N} \to \mathbb{P}.\ \mathcal{E}p \wedge \neg \mathcal{E}\overline{p} \wedge \neg \mathcal{D}p$

Proof. Let θ be given as in EPF. Define $\mathcal{K}c := \exists v. \theta_c c \triangleright v$. $\mathcal{K}$ is semi-decided by $\lambda c n.$**if** $\mathsf{seval}(\theta_c c) n$ **is** $\mathsf{Some}\, v$ **then** true **else** false and thus enumerable by Corollary 4.55.

We prove that $\overline{\mathcal{K}}$ is not semi-decidable, yielding both $\neg \mathcal{E}\overline{\mathcal{K}}$ and $\neg \mathcal{D}\mathcal{K}$ by Lemmas 4.55 and 4.48. Let $\overline{\mathcal{K}}$ be semi-decidable, i.e. by Fact 4.63 there is $f : \mathbb{N} \rightharpoonup Y$ s.t. $\neg \mathcal{K} x \leftrightarrow \exists y.\ fx \triangleright y$. Define $f' : \mathbb{N} \rightharpoonup \mathbb{B}$ as $f' x := fx \gg\!= \lambda\_.\ \mathsf{ret}\,\mathsf{true}$. Now f' has a code c s.t. $\forall x.\ ecx \triangleright f' x$ by universality of θ.

We have a contradiction via

$$\neg \mathcal{K}c \leftrightarrow (\exists y.\, f\, c \triangleright y) \leftrightarrow (\exists y.\, f' x \triangleright y)(\exists y. ecc \triangleright y) \leftrightarrow \mathcal{K}c. \qquad \blacksquare$$

6.4 The Enumerability Axiom

Using EPF or SCT as basis for synthetic computability requires the manipulation of partial functions or stationary functions, which is tedious. Alternatively, synthetic computability can be presented even more elegantly by an equivalent axiom concerned with enumerable predicates rather than partial functions. A non-parametric enumerability axiom is used by Bauer [10] together with countable choice to develop synthetic computability results.

We introduce the **parametric enumerability axiom** postulating an enumerator $\varphi : \mathbb{N} \to (\mathbb{N} \to \mathbb{ON})$ which is parametrically universal for all parametrically enumerable predicates $p : \mathbb{N} \to \mathbb{N} \to \mathbb{P}$:

Def. parametric enumerability axiom

$$\mathsf{EA} := \Sigma \varphi : \mathbb{N} \to (\mathbb{N} \to \mathbb{ON}). \forall (p : \mathbb{N} \to \mathbb{N} \to \mathbb{P}).$$

$$(\exists (f : \mathbb{N} \to \mathbb{N} \to \mathbb{ON}). \forall i.\, f_i \textit{ enumerates } p_i) \to \exists \gamma : \mathbb{N} \to \mathbb{N}. \forall i.\, \varphi_{\gamma i} \textit{ enumerates } p_i$$

That is, EA states that whenever p is parametrically enumerable, then $\lambda i.\, \varphi_{\gamma i}$ parametrically enumerates p for some γ.

parametric enumerability
→ Sec. 4.2, Page 34

Note the two different roles of natural numbers in the axiom: If we would consider predicates over a general type X we would have $\varphi : \mathbb{N} \to (\mathbb{N} \to \mathbb{O}X)$.

Equivalently, we could have required that p is enumerable:

Lemma 6.10. $\mathsf{EA} \leftrightarrow \Sigma \varphi.\, \forall p : \mathbb{N} \to \mathbb{N} \to \mathbb{P}.\, \mathcal{E}(\lambda(x, y).\, p(x, y)) \to \exists \gamma : \mathbb{N} \to \mathbb{N}. \forall i.\, \varphi_{\gamma i}$ *enumerates* pi

Proof. Immediate by Corollary 4.24. $\blacksquare$

Again equivalently, EA can be stated to only mention enumerators instead of predicates, which is the formulation of EA used in [70].

Fact 6.11. $\mathsf{EA} \leftrightarrow \Sigma \varphi : \mathbb{N} \to (\mathbb{N} \to \mathbb{ON}). \forall f : \mathbb{N} \to \mathbb{N} \to \mathbb{ON}. \exists \gamma. \forall x.\, \varphi_{\gamma x} \equiv_{\mathsf{ran}} f\, x$

In this formulation, φ is a surjection w.r.t. range equivalence $f \equiv_{\mathsf{ran}} g$, where $\varphi_c \equiv_{\mathsf{ran}} f \leftrightarrow \forall x. (\exists n. \varphi_c n = \mathsf{Some}\, x) \leftrightarrow (\exists n. f\, n = \mathsf{Some}\, x)$.

Given φ, we define $\mathcal{W}_c x := \exists n.\, \varphi_c n = \mathsf{Some}\, x$ and the problem $\mathcal{K}$ as the diagonal of $\mathcal{W}$, i.e. $\mathcal{K}c := \mathcal{W}_c c$. We call $\mathcal{W}$ a **universal table**. In Section 8.1 we will show that $\mathcal{W}$ and $\mathcal{K}$ are m-equivalent, and both are m-complete. For now we only use $\mathcal{K}$ to note the following result:

Def. universal table

Lemma 6.12. $\mathsf{EA} \to \Sigma p : \mathbb{N} \to \mathbb{P}.\, \mathcal{E}p \wedge \neg \mathcal{E}\overline{p} \wedge \neg \mathcal{D}p$

Proof. We pick p as $\mathcal{K}c := \mathcal{W}_c c$. $\mathcal{K}$ is enumerated by $\lambda \langle c, m \rangle$. **if** $\varphi_c m$ **is** $\mathsf{Some}\, x$ **then if** $x =_{\mathbb{B}} c$ **then** $\mathsf{Some}\, c$ **else** None **else** None. If $\overline{\mathcal{K}}$ would be enumerable, there would be a code c s.t. $\forall x.\, \mathcal{W}_c x \leftrightarrow \overline{\mathcal{K}} x$. In particular $\mathcal{W}_c c \leftrightarrow \overline{\mathcal{K}} c \leftrightarrow \neg \mathcal{W}_c c$. $\blacksquare$

Similarly to how SCT can be reformulated by letting ϕ be universal for unary functions and introducing an explicit S_1^1-operator, EA can also be stated in this fashion, with an S_1^1-operator w.r.t. $\mathcal{W}$.

Lemma 6.13. $\mathsf{EA} \leftrightarrow \Sigma \varphi. (\forall p. \mathcal{E}p \to \exists c. \varphi_c$ *enumerates* $p) \wedge \Sigma \sigma : \mathbb{N} \to \mathbb{N} \to \mathbb{N}. \forall c x y. \mathcal{W}_{(\sigma c x)} y \leftrightarrow \mathcal{W}_c \langle x, y \rangle$

Proof. The direction from right to left is straightforward using Lemma 6.10.

For the direction from left to right, let φ be given.

For the first part of the conclusion let p be given and enumerable. Then $\lambda xy.\ py$ is parametrically enumerable, so let γ be given from EA. Then $\varphi_{\gamma 0}$ enumerates p.

For the second part, let $p\langle c,x\rangle y := \exists n.\varphi_c n = \mathsf{Some}\,\langle x,y\rangle$. Since p is enumerable, by Lemma 6.10 and EA there is γ s.t.$\varphi_{\gamma\langle c,x\rangle}$ enumerates $p\langle c,x\rangle$. Now $Scx := \gamma\langle c,x\rangle$ is the wanted function. ∎

SCT and EA are equivalent. For the forwards direction, we show that enumerators $\mathbb{N}\to\mathbb{ON}$ can be equivalently given as functions $\mathbb{N}\to\mathbb{N}$.

Theorem 6.14. SCT → EA

Proof. Let a universal function ϕ be given. Define:

$$\varphi_c\langle n,m\rangle := \textbf{if } \phi_c^n m \textbf{ is } \mathsf{Some}\,(\mathsf{S}\,x) \textbf{ then } \mathsf{Some}\,x \textbf{ else } \mathsf{None}$$

Let $f:\mathbb{N}\to\mathbb{N}\to\mathbb{ON}$ be a parametric enumerator for p. We define $f':\mathbb{N}\to\mathbb{N}\to\mathbb{N}$ as $f'xn :=$ **if** $f\,xn$ **is** $\mathsf{Some}\,y$ **then** $\mathsf{S}\,y$ **else** 0. By SCT, we obtain a function γ for f', and we have

$$\begin{aligned} pxy &\leftrightarrow \exists n.\ f\,xn = \mathsf{Some}\,y \\ &\leftrightarrow \exists n.\ f'xn = \mathsf{S}\,y \\ &\leftrightarrow \exists nm.\ \phi_{\gamma x}^n m = \mathsf{Some}\,(\mathsf{S}\,y) \\ &\leftrightarrow \exists nm.\ \varphi_{\gamma x}\langle n,m\rangle = \mathsf{Some}\,y \\ &\leftrightarrow \exists k.\ \varphi_{\gamma x} k = \mathsf{Some}\,y \end{aligned}$$

For the converse direction, we use that the graph of functions is enumerable.

Theorem 6.15. EA → SCT

Proof. Let φ as in EA be given. Recall $\mathsf{mkstat}:(\mathbb{N}\to\mathbb{O}X)\to\mathbb{N}\to\mathbb{O}X$ from Section 4.5.3 turning arbitrary $F:\mathbb{N}\to\mathbb{ON}$ into stationary sequences. We define

$$\varphi_c^n x := \mathsf{mkstat}(\lambda n.\ \textbf{if } \varphi_c n \textbf{ is } \mathsf{Some}\,\langle x',y\rangle \textbf{ then if } x' =_{\mathbb{B}} x \textbf{ then } \mathsf{Some}\,y \textbf{ else } \mathsf{None} \textbf{ else } \mathsf{None})n$$

Let $f:\mathbb{N}\to\mathbb{N}\to\mathbb{N}$ and let $\varphi_{\gamma x}$ enumerate $\lambda x\langle n,m\rangle.\ f\,xn = m$ via EA. Now γ serves as coding function for f by Fact 4.69. ∎

6.5 Rice's theorem

One of the central results of every introduction to computability theory is Rice's theorem [197], stating that non-trivial semantic predicates on programs are undecidable. Two proof strategies can be found in the literature: By using a fixed-point theorem or by establishing a many-one reduction from $\overline{\mathcal{K}}$. We here give synthetic variants of both proofs.

[197] Rice. 1953. Classes of recursively enumerable sets and their decision problems.

We base the first proof on the axiom EPF, since the notion of a fixed-point is more natural there. We base the second proof on the axiom EA. Here the choice is less canonical, but using EA enables a comparison of EA and EPF as axioms for synthetic computability.

We start by assuming EPF and proving a fixed-point theorem due to Rogers [202].

Theorem 6.16. Let θ be given as in EPF and $\gamma : \mathbb{N} \to \mathbb{N}$, then there exists c such that $\theta_{\gamma c} \equiv \theta_c$.

Proof. Let $\gamma : \mathbb{N} \to \mathbb{N}$. Let $f_x z := \theta_x x \gg\!= \lambda y.\theta_y z$ and γ' via EPF be such that $\theta_{\gamma' x} \equiv f_x$ (1). Let c via EPF be such that $\forall x.\ \theta_c x \triangleright \gamma(\gamma' x)$ (2).

Now $f_c \equiv \theta_{\gamma' c}$ by (1).

Also $f_c z \equiv (\theta_c c \gg\!= \lambda y.\theta_y z) \equiv \theta_{\gamma(\gamma' x)} z$ by the definition of f and (2).

Now $\gamma' c$ is a fixed-point for $\lambda i.\theta_{\gamma i}$: $\theta_{\gamma(\gamma' c)} \equiv f_c \equiv \theta_{\gamma' c}$. ∎

Rice's theorem can then be stated and proved as follows:

Theorem 6.17. Let θ be given as in EPF and $p : \mathbb{N} \to \mathbb{P}$. If p treats elements as codes w.r.t. θ and is non-trivial, then p is undecidable. Formally:

$$(\forall cc'.\ \theta_c \equiv \theta_{c'} \to pc \leftrightarrow pc') \to \forall c_1 c_2.\ pc_1 \to \neg pc_2 \to \neg \mathcal{D} p$$

Proof. Let f decide p and let pc_1 and $\neg pc_2$. Define $h : \mathbb{N} \to \mathbb{N} \rightharpoonup \mathbb{N}$ as $h_x := \textbf{if } f\,x \textbf{ then } \theta_{c_2} \textbf{ else } \theta_{c_1}$ and let γ via EPF be such that $\theta_{\gamma x} y \equiv h_x y$. Let c be a fixed-point for γ via Theorem 6.16, i.e. $\theta_{\gamma c} \equiv \theta_c$.

Then either $f\,c = \mathsf{true}$ and thus pc, but $\theta_c \equiv \theta_{c_2}$ and thus pc_2. A contradiction.

Or $f\,c = \mathsf{false}$ and thus $\neg pc$, but $\theta_c \equiv \theta_{c_1}$ and thus $\neg pc_1$. A contradiction. ∎

Rice's theorem is often also stated for predicates $p : (\mathbb{N} \rightharpoonup \mathbb{N}) \to \mathbb{N}$. This formulation has the advantage that the requirement on p does not have to mention θ.

Corollary 6.18. EPF implies that if $p : (\mathbb{N} \rightharpoonup \mathbb{N}) \to \mathbb{P}$ is extensional and non-trivial, then p is undecidable. Formally:

$$\mathsf{EPF} \to (\forall f f' : \mathbb{N} \rightharpoonup \mathbb{N}.\, f \equiv_{\mathbb{N} \to \mathbb{N}} f' \to pf \leftrightarrow pf') \to \forall f_1 f_2.\ pf_1 \to \neg pf_2 \to \neg \mathcal{D} p$$

Proof. Let p be decidable. We define the index predicate of p as $I_p := \lambda c : \mathbb{N}.\ p(\theta_c)$, and have $I_p \preceq_m p$. Thus since p is decidable, I_p is decidable. Since I_p treats elements as codes and is non-trivial using EPF, we have that I_p is undecidable by Theorem 6.17. Contradiction. ∎

A second proof strategy for Rice's theorem is by establishing a many-one reduction from a problem proved undecidable via diagonalisation. We could use $\mathcal{K}$ defined using EPF in Fact 6.9, but here use EA to compare the two axioms. Thus, we use the problem $\mathcal{K}$ as used in Lemma 6.12. We follow Forster and Smolka [83], who mechanise a fully constructive proof of Rice's theorem based on the call-by-value λ-calculus by isolating a reduction lemma ("Rice's Lemma").

[83] Forster and Smolka. 2017. Weak Call-by-Value Lambda Calculus as a Model of Computation in Coq.

Lemma 6.19. Let φ be given as in EA and $p : \mathbb{N} \to \mathbb{P}$. If p treats elements as codes w.r.t. φ, p is non-trivial and $pc_\emptyset$ for $c_\emptyset$ being a code for the empty predicate, then $\overline{\mathcal{K}} \preceq_m p$.

Formally let $\mathcal{W}_c x := \exists n.\ \varphi_c n = \mathsf{Some}\, x$ be the universal table for φ. We then have:

$$(\forall cc'.\ (\forall x.\ \mathcal{W}_c x \leftrightarrow \mathcal{W}_{c'} x) \to pc \leftrightarrow pc') \to \forall c_\emptyset c_0.\ (\forall x. \neg \mathcal{W}_{c_\emptyset} x) \to pc_\emptyset \to \neg pc_0 \to \overline{\mathcal{K}} \preceq_m p$$

Proof. The predicate $q := \lambda x y.\ \mathcal{K}x \wedge \mathcal{W}_{c_0} y$ is enumerable by Corollary 4.18, meaning we obtain γ from EA s.t. $\forall x y.\ \mathcal{W}_{\gamma x} y \leftrightarrow \mathcal{K}x \wedge \mathcal{W}_{c_0} y$.

Let $\neg \mathcal{K}x$. We have $\mathcal{W}_{\gamma x} y \leftrightarrow \bot \leftrightarrow \mathcal{W}_{c_\emptyset} y$. Since $pc_\emptyset$ and p is semantic also $p(\gamma x)$.

Conversely, let $p(\gamma x)$ and $\mathcal{K}x$. We have $\mathcal{W}_{\gamma x} y \leftrightarrow \mathcal{W}_{c_0} y$. Since p is semantic, also pc_0. Contradiction. ∎

Theorem 6.20. Let φ be given as in EA and $p : \mathbb{N} \to \mathbb{P}$. If p treats inputs as codes w.r.t. φ and p is non-trivial, then p is not bi-enumerable.

Formally let $\mathcal{W}_c x := \exists n.\ \varphi_c n = \mathsf{Some}\, x$ be the universal table for φ. We then have:

$$(\forall cc'.(\forall x.\ \mathcal{W}_c x \leftrightarrow \mathcal{W}_{c'} x) \to pc \leftrightarrow pc') \to \forall c_1 c_2.\ pc_1 \to \neg pc_2 \to \neg(\mathcal{E}p \wedge \mathcal{E}\overline{p})$$

Proof. Since $\lambda x : \mathbb{N}.\bot$ is enumerable, by EA there is $c_\emptyset$ s.t. $\forall x.\neg W_{c_\emptyset} x$. Now let pc_1, $\neg pc_2$, and let p be bi-enumerable.

If $pc_\emptyset$, we have $\overline{\mathcal{K}} \preceq_m p$, a contradiction since $\overline{\mathcal{K}}$ would be enumerable by Lemma 5.4.

If $\neg pc_\emptyset$ we have $\overline{\mathcal{K}} \preceq_m \overline{p}$, again a contradiction. ∎

Corollary 6.21. Let φ be given as in EA and $p : \mathbb{N} \to \mathbb{P}$. If p treats inputs as codes w.r.t. φ and p is non-trivial, then p is undecidable.

We can state this second version of Rice's theorem for $p : (\mathbb{N} \to \mathbb{P}) \to \mathbb{P}$.

Corollary 6.22. EA implies that if p is extensional and non-trivial w.r.t. enumerable predicates, then p is undecidable. Formally:

$$\mathsf{EA} \to (\forall qq' : \mathbb{N} \to \mathbb{P}.(\forall x.\ qx \leftrightarrow q'x) \to pq \leftrightarrow pq') \to \forall q_1 q_2.\ \mathcal{E}q_1 \to \mathcal{E}q_1 \to pq_1 \to \neg pq_2 \to \neg \mathcal{D}p$$

Proof. Let p be decidable. We define the index predicate of p as $I_p := \lambda c : \mathbb{N}.\ p(\mathcal{W}_c)$, and have $I_p \preceq_m p$. Thus since p is decidable, I_p is decidable. Since I_p treats elements as codes and is non-trivial using EA, we have that I_p is undecidable by Corollary 6.21. Contradiction. ∎

We have formulated both theorems to explicitly assume θ and φ and their respective specification, to contrast the axioms EPF and EA. One can however obtain Theorem 6.17 from Corollary 6.21 – and vice versa – constructing θ from φ and constructing a predicate q treating elements as indices w.r.t. θ from a predicate p treating elements as indices w.r.t. φ – and vice versa.

Proofs based on EPF require the manipulation of partial functions, which is formally tedious. We will thus use EA as basis for synthetic computability: In contrast to SCT, it does not force us to encode every computation as total function $\mathbb{N} \to \mathbb{N}$, and in contrast to EPF it does not force us to work with partial functions either.

Instead, we can simply consider enumerable predicates (with many closure properties from Chapter 4) and their enumerators (which are total functions).

6.6 Related work

We have discussed the history of CT and synthetic computability in Chapter 2.

When we write CT without an index, we mean CT_{ϕ_μ}, i.e. the version of CT stating that a step-indexed interpreter is universal for $\mathbb{N} \to \mathbb{N}$, or equivalently that every function of type $\mathbb{N} \to \mathbb{N}$ is μ-recursively computable. This axiom was introduced by Kreisel [142], and is discussed extensively e.g. by Troelstra and van Dalen [228]. We discuss a mechanised definition of CT in Chapter 29, and its consistency for CIC in Section 6.1.

[142] Kreisel. 1965. Mathematical logic.

[228] Troelstra and van Dalen. 1988. Constructivism in mathematics. Vol. I.

[10] Bauer. 2006a. First steps in synthetic computability theory.

We here in detail compare our axioms with the ones used by Bauer [10] and the ones used by Richman [198] and Bridges and Richman [25].

Bauer [10] develops synthetic computability based on an axiom stating that the set of enumerable sets of natural numbers is enumerable. Translating to our type theoretic setting this yields the following axiom:

[10] Bauer. 2006a. First steps in synthetic computability theory.

$$\mathsf{EA}' := \exists \mathcal{W}\colon \mathbb{N} \to (\mathbb{N} \to \mathbb{P}). \forall p\colon \mathbb{N} \to \mathbb{P}.\ \mathcal{E}p \leftrightarrow \exists c.\ \mathcal{W}_c \equiv_{\mathbb{N}\to\mathbb{P}} p$$

That is, EA' states that there is an enumerator $\mathcal{W}$ of all enumerable predicates, up to extensionality.

Additionally to EA', Bauer also assumes countable choice and Markov's principle. In general however, the assumption of countable choice makes the theory anti-classical, i.e. assuming LEM is inconsistent. We discuss this interplay between axioms like EA', MP, LEM, and countable choice in detail in Chapter 7. Countable choice allows extracting the enumerator for every enumerable predicate in the range of $\mathcal{W}$ computationally, corresponding to a non-parametric version of our axiom EA. Countable choice also can be used to prove a synthetic S_n^m theorem w.r.t. $\mathcal{W}$, but apart from those two applications is not needed.

Our parametric formulation of EA implies EA', and conversely EA' implies EA under the presence of countable choice. We only prove the former:

Theorem 6.23. $\mathsf{EA} \to \mathsf{EA}'$

Proof. Let a universal enumerator φ be given. Define $\mathcal{W}_c x := \exists n.\varphi_c n = \mathsf{Some}\, x$. Since we have that φ_c enumerates p if and only if $\mathcal{W}_c \equiv_{\mathbb{N}\to\mathbb{P}} p$ we have $\mathcal{E}p \to \exists c.\ \mathcal{W}_c \equiv_{\mathbb{N}\to\mathbb{P}} p$.

Vice versa if $\mathcal{W}_c \equiv_{\mathbb{N}\to\mathbb{P}} p$, φ_c enumerates p and thus $\mathcal{E}p$. ∎

[198] Richman. 1983. Church's thesis without tears.

Richman [198] introduces the axiom CPF ("Countability of Partial Functions"). It states that the set of partial functions is (extensionally) countable, i.e. there is a surjection $\mathbb{N} \to (\mathbb{N} \rightharpoonup \mathbb{N})$ w.r.t. equivalence on partial functions. Intensionally, Richman models the partial function space $\mathbb{N} \rightharpoonup \mathbb{N}$ as stationary functions as discussed in Section 4.5. Thus, written out fully his axiom is a non-parametric version of EPF, just instantiated to the stationary functions model of partial functions.

[25] Bridges and Richman. 1987. Varieties of constructive mathematics.

Theory based on CPF is developed in the book by Bridges and Richman [25], where CPF is taken as basis for the constructivist system RUSS. In RUSS, the axiom of countable choice is also assumed. Bridges and Richman discuss that "in RUSS countable choice can usually be avoided" [25, p. 54] by postulating a composition operator for θ or, equivalently, an SMN operator. The theory we develop in Chapter 8 is in strong support of this conjecture.

In this chapter we have also given two proof of Rice's theorem. The first proof is based on a parametrically universal partial function θ, while the second proof is based on a parametrically universal enumerator φ. The two proofs of Rice's theorem use different proof strategies.

The second strategy establishes a reduction from $\mathcal{K}$. This strategy is used in the textbooks by Cutland [44], Odifreddi [180], Soare [210], and Cooper [36], whereas Rogers [202] and Sipser [207] pose Rice's theorem as an exercise.

The first strategy, based on Rogers' fixed-point theorem or equivalently Kleene's recursion theorem is less frequently found. It is however mentioned in the Wikipedia article on Rice's theorem [236]. The technique appears first in the lecture notes by Scott [204], who shows a variant of Rice's theorem for the λ-calculus. Scott's proof can also be found in [209, 8].

[204] Scott. 1968. A system of functional abstraction.

[178] Norrish. 2011. Mechanised Computability Theory.

We are aware of five machine-checked proofs of Rice's theorem in the literature: Norrish [178] proves Rice's theorem for the λ-calculus, formulated for predicates $p\colon(\mathbb{N}\to\mathbb{ON})\to\mathbb{P}$, using the proof strategy via reduction. Forster and Smolka [83] prove Rice's theorem for the weak call-by-value λ-calculus, formulated for predicates on terms of the considered calculus which have the same extensional behaviour, using the proof strategy via reduction. Forster [69] proves Scott's variant of Rice's theorem for the weak call-by-value λ-calculus, formulated for predicates on terms of the considered calculus which do not distinguish β-equivalent terms, using a fixed-point theorem. Carneiro [26] proves Rice's theorem for μ-recursive functions, formulated for predicates $p\colon(\mathbb{N}\rightharpoonup\mathbb{N})\to\mathbb{P}$, using a fixed-point theorem. Ferreira Ramos, Almeida, and Ayala-Rincón [67] prove Rice's theorem for the functional language PVS0, formulated for predicates on PVS0, using an assumed fixed-point theorem for PVS0.

[83] Forster and Smolka. 2017. Weak Call-by-Value Lambda Calculus as a Model of Computation in Coq.

[69] Forster. 2014. A Formal and Constructive Theory of Computation.

[26] Carneiro. 2019. Formalizing Computability Theory via Partial Recursive Functions.

[67] Ferreira Ramos et al.. 2020. Formalization of Rice's Theorem over a Functional Language Model.

Bauer [10] also presents a synthetic variant of Rice's theorem. His formulation reads "If A is a set such that all functions of type $A\to A$ have a fixed-point, every function $A\to\mathbb{B}$ is constant" and uses the enumerability axiom as discussed above, but does not rely on countable choice to the best of our knowledge. Note that our variants of Rice's theorem presented in this thesis are trivialities in classical set theory, the foundation of textbook computability, since both EPF and EA are false in classical set theory where all problems have a characteristing decision function. In contrast, Bauer's theorem is a triviality in classical set theory in *two* ways: First, the enumerability axiom is contradictory in classical set theory, and second the statement of the theorem is a trivial even without axioms since if all functions $A\to A$ have a fixed-point, A is a sub-singleton: two distinct elements a_1, a_2 would allow constructing a fixed-point free function $\lambda x.\mathbf{if}\ x = a_1\ \mathbf{then}\ a_2\ \mathbf{else}\ a_1$.

We thus sacrifice identifying the minimal essence of theorems to better preserve classical intuitions.

CHAPTER 7

Axioms in relation to synthetic computability

In the previous chapter we have introduced synthetic, parametric variants of CT. It is easy to see that both CT and its synthetic variants are in conflict with traditional classical mathematics, since the law of excluded middle LEM together with a form of the axiom of countable choice $\mathsf{AC}_{\mathbb{N},\mathbb{N}}$ allows the definition of non-computable functions [228]. This observation can be sharpened in various ways: To define a non-computable function directly, the weak limited principle of omniscience WLPO and the countable unique choice axiom $\mathsf{AUC}_{\mathbb{N},\mathbb{B}}$ suffice. Alternatively, Kleene noticed that there is a decidable tree predicate with infinitely many nodes but no computable infinite path [135]. If functions and computable functions are identified via SCT, a Kleene tree is in conflict with weak Kőnig's lemma WKL and with Brouwer's fan theorem.

[228] Troelstra and van Dalen. 1988. Constructivism in mathematics. Vol. I.

It is however well-known that CT is consistent in Heyting arithmetic with Markov's principle MP [133] which given CT states that termination of computation is stable under double negation. Recently, Swan and Uemura [220] proved that CT is consistent in univalent Martin-Löf type theory with propositional truncation and MP, and Yamada [240] proved that a formulation of CT in predicative intensional Martin-Löf type theory is consistent.

[133] Kleene. 1945. On the interpretation of intuitionistic number theory.

[220] Swan and Uemura. 2019. On Church's Thesis in Cubical Assemblies.

[240] Yamada. 2020. Game semantics of Martin-Löf type theory, part III: its consistency with Church's thesis.

While predicative Martin-Löf type theory as formalisation of Bishop's constructive mathematics proves full axiom of choice AC (since $\exists$ is modelled as Σ), univalent type theory usually only proves the axiom of unique choice AUC (since there is a notion of homotopy propositions to define $\exists$). But since $\mathsf{AUC}_{\mathbb{N},\mathbb{B}}$ suffices to show that LEM implies $\neg$CT, classical logical axioms are incompatible with CT in both predicative and in univalent type theory.[1]

In CIC neither AC, nor AUC or $\mathsf{AUC}_{\mathbb{N},\mathbb{B}}$ are provable, as discussed in Section 7.6. However, choice axioms as well as LEM can be consistently assumed in CIC [235]. Furthermore, it seems likely that the consistency proof for CT in [220] can be adapted for CIC.

[235] Werner. 1997. Sets in types, types in sets.

It seems like to disprove CT in CIC one needs a (weak) classical logical axiom to perform logical decisions for non-decidable predicates in proofs, and a (weak, non-extensional) choice axiom to turn total functional relations into functions subject to CT. This puts CIC in a special position: Since it proves no classical logical axioms and virtually no choice axioms, assuming just classical logical axioms or just choice axioms might be consistent with CT. This chapter is intended to serve as a preliminary report towards this consistency question, approximating it by surveying results from intuitionistic logic and constructive reverse mathematics in constructive type theory with a separate universe of propositions, with a special focus on CT and other axioms based on notions from computability theory. Specifically, we discuss the following propositional axioms:

- Kleene trees (KT) in Section 7.2

[1] For this chapter, we reserve the term "classical logical axioms" for axioms implying or implied by LEM. In particular, non-extensional versions of the axiom of choice are not covered by this wording.

- extensionality axioms like functional extensionality (Fext), propositional extensionality (Pext), and proof irrelevance (PI) in Section 7.3
- classical logical axioms like the principle of excluded middle (LEM, WLEM), independence of premises (IP), and limited principles of omniscience (LPO, WLPO, LLPO) in Section 7.4
- axioms of Russian constructivism like Markov's principle (MP) in Section 7.5
- choice axioms like the axiom of choice (AC), countable choice (ACC, $\mathsf{AC}_{\mathbb{N},\mathbb{N}}$, $\mathsf{AC}_{\mathbb{N},\mathbb{B}}$), dependent choice (ADC), and unique choice (AUC, $\mathsf{AUC}_{\mathbb{N},\mathbb{B}}$) in Section 7.6
- axioms on trees like weak Kőnig's lemma (WKL) and the fan theorem (FAN) in Section 7.7
- axioms regarding continuity and Brouwerian principles (Homeo, Cont, WC-N) in Section 7.8

The following hyper-linked diagram displays provable implications and incompatible axioms.

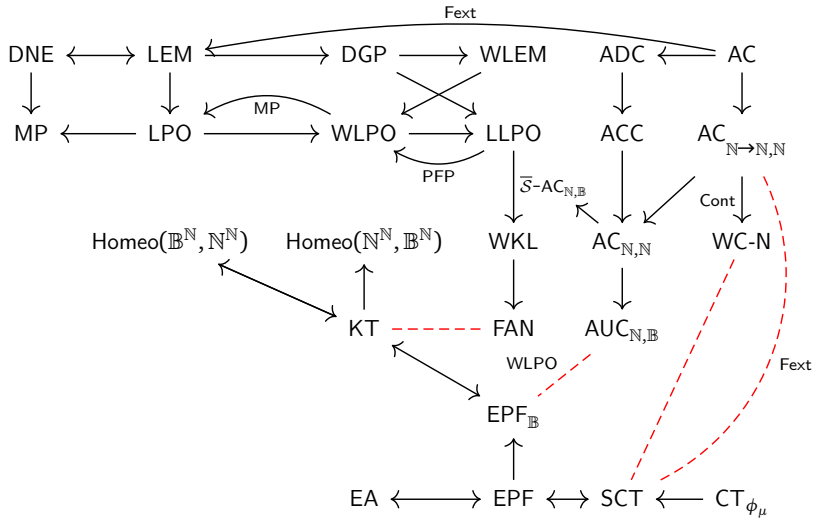


Figure 7.1.: Overview of results. → are implications, ---- denotes incompatible axioms.

Outline Section 7.1 recaps the constructivist axiom CT and its synthetic variant EPF. Section 7.2 introduces decidable binary trees and constructs a Kleene tree. The connection of CT to the classes of axioms as listed above is surveyed in Sections 7.3 to 7.8. Section 7.9 contains concluding remarks.

Publications Many pieces of text of this chapter are adapted from [70].

[70] Forster. "Church's thesis and related axioms in Coq's type theory." *29th EACSL Annual Conference on Computer Science Logic (CSL 2021)*.

7.1 Consistency and admissibility of CT

The axiom CT states that every total function $f : \mathbb{N} \to \mathbb{N}$ is computable in a fixed, Turing-complete model of computation:

$$\mathsf{CT} := \forall f : \mathbb{N} \to \mathbb{N}. \exists c : \mathbb{N}. \forall x : \mathbb{N}. \exists n : \mathbb{N}.\ \phi_c^n x = \mathsf{Some}(f\,x)$$

Here, ϕ is a step-indexed interpreter for μ-recursive functions, i.e. $\phi_c^n x$ evaluates the c-th μ-recursive function on input x for n steps. Recall that in Section 6.1 we have discussed the axiom CT_ϕ for arbitrary interpreters ϕ.

[132] Kleene. 1943. Recursive predicates and quantifiers.

In 1943, Kleene conjectured that whenever $\forall x.\exists y.\, Rxy$ is constructively provable, there in fact exists a μ-recursive function f such that $\forall x.\, Rx(fx)$ [132]. This corresponds to a strong form of the admissibility of CT. In 1945, Kleene [133] proved his conjecture for Heyting arithmetic, using number realizability. An independent proof of this is due to Beth [21].

[133] Kleene. 1945. On the interpretation of intuitionistic number theory.

[21] Beth. 1948. Semantical Considerations on Intuitionistic Mathematics.

Troelstra and van Dalen [228, §4.5.1 p. 204] state an even stronger result, using Gödel's Dialectia interpretation [95], namely that in Heyting arithmetic CT, MP and a restricted form of the independence of premise rule IP (with P logically decidable, see Section 7.4) are consistent as schemes.

[228] Troelstra and van Dalen. 1988. Constructivism in mathematics. Vol. I.

[95] Gödel. 1958. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes.

Odifreddi states that "for all current intuitionistic systems (not involving the concept of choice sequence) the consistency with CT has actually been established" [180, §1.8 pg. 122]. We do not discuss other systems for constructive or intuitionistic mathematics in detail.

[180] Odifreddi. 1992. Classical recursion theory: The theory of functions and sets of natural numbers.

For CIC, the result is not explicitly stated in the literature. An admissibility proof of CT seems to be immediate as a consequence of Letouzey's semantics extraction theorem for Coq [156], we discuss this in more detail in Chapter 29. Regarding a consistency proof one cannot mirror the situation in Heyting arithmetic, since a Dialectia interpretation for Coq is not available [183].

[156] Letouzey. 2004. Programmation fonctionnelle certifiée: l'extraction de programmes dans l'assistant Coq.

[183] Pédrot. 2015. A Materialist Dialectica.

However, several approaches seem to yield the result:

First, CT is consistent in intuitionistic set theory (e.g. IZF) [102], and IZF can be used to model CIC [9].

[102] Hahanyan. 1981. The consistency of some intuitionistic and constructive principles with a set theory.

[9] Barras. 2010. Sets in Coq, Coq in Sets.

Secondly, realizability models based on the first Kleene algebra prove CT consistent. Luo constructs an ω-set model for the Extended Calculus of Constructions (ECC, a type theory with type universes and impredicative $\mathbb{P}$, but no inductive types), where "[t]he morphisms between ω-sets are 'computable' in the sense that they are realised by partial recursive functions" [161, §6.1 pg. 118].

[161] Luo. 1994. Computation and Reasoning: A Type Theory for Computer Science.

Thirdly, it is well known how to build topos models of the calculus of constructions [120]. The effective topos, due to Hyland [119], validates CT.

[120] Hyland and Pitts. 1989. The Theory of Constructions: Categorical Semantics and Topos-theoretic Models.

Fourthly, Swan and Uemura [220] give a sheaf model construction proving that CT is consistent in Martin Löf type theory, together with propostional truncation, Markov's principle, and univalence. It seems like the syntactic universe of propositions $\mathbb{P}$ does not hinder adapting their model construction to CIC.

[220] Swan and Uemura. 2019. On Church's Thesis in Cubical Assemblies.

Fivthly, Yamada [240] gives a game-semantic proof that a $\forall f.\Sigma c$ form of CT is consistent in intensional Martin Löf type theory, settling an open question of at least 15 years [123]. Note that this form is significantly stronger, since it allows defining a strictly intensional higher-order coding *function* of type $(\mathbb{N}\to\mathbb{N})\to\mathbb{N}$, which is inconsistent under the assumption of functional extensionality. We discuss this in Section 7.3. It is not obvious how to extend Yamada's proof to our $\forall f.\exists c$ formulation of CT in CIC with the impredicative universe $\mathbb{P}$.

[240] Yamada. 2020. Game semantics of Martin-Löf type theory, part III: its consistency with Church's thesis.

[123] Ishihara, Maietti, Maschio, and Streicher. 2018. Consistency of the intensional level of the Minimalist Foundation with Church's thesis and axiom of choice.

We have discussed SCT as fully synthetic variant of CT in Chapter 6. All variants of SCT we discussed are consequences of CT, and thus similarly consistent. For this chapter, we recall SCT and the equivalent variant $\mathsf{EPF}_{\mathbb{B}}$:

$$\mathsf{SCT} := \Sigma\phi\colon \mathbb{N}\to\mathbb{N}\to\mathbb{N}\to\mathbb{ON}.$$
$$(\forall cxn_1n_2v.\ \phi_c^{n_1}x = \mathsf{Some}\,v \to n_2 \geq n_1 \to \phi_c^{n_2} = \mathsf{Some}\,v) \wedge$$
$$\forall(f\colon\mathbb{N}\to\mathbb{N}\to\mathbb{N}).\ \exists\gamma\colon\mathbb{N}\to\mathbb{N}.\ \forall ix.\exists n.\ \phi_{\gamma i}^{n}x = \mathsf{Some}\,(f_i x)$$
$$\mathsf{EPF}_{\mathbb{B}} := \Sigma\theta\colon\mathbb{N}\to(\mathbb{N}\rightharpoonup\mathbb{B}).\forall(f\colon\mathbb{N}\to\mathbb{N}\rightharpoonup\mathbb{B}).\exists\gamma\colon\mathbb{N}\to\mathbb{N}.\forall i.\ \theta_{\gamma i}\equiv_{\mathbb{N}\to\mathbb{B}} f_i$$

Fact 7.1. $\mathsf{EPF}_{\mathbb{B}} \vee \mathsf{SCT} \to \exists p\colon\mathbb{N}\to\mathbb{P}.\ \mathcal{E}p \wedge \neg\mathcal{E}\overline{p} \wedge \neg\mathcal{D}p \wedge \neg\mathcal{D}\overline{p}$

Note that SCT and $\mathsf{EPF}_{\mathbb{B}}$ as we defined them are no propositions, but can be made propositions by existentially quantifying ϕ and θ respectively. This difference however seems to have no consequences for consistency or compatibility with other axioms.

Even without axioms, we can define two fully synthetic problems where the definition does not rely on axioms:

$$\mathcal{K}_{\mathbb{N}\to\mathbb{B}}(f\colon\mathbb{N}\to\mathbb{B}) := \exists n.\ fn = \mathsf{true} \qquad \mathcal{K}_{\mathbb{N}\to\mathbb{N}}(f\colon\mathbb{N}\to\mathbb{N}) := \exists n.\ fn \neq 0$$

Fact 7.2. $\mathcal{S}p \leftrightarrow p \preceq_m \mathcal{K}_{\mathbb{N}\to\mathbb{B}}$

Lemma 7.3. $\mathcal{K}_{\mathbb{N}\to\mathbb{B}} \equiv_m \mathcal{K}_{\mathbb{N}\to\mathbb{N}}$, $\overline{\mathcal{K}_{\mathbb{N}\to\mathbb{N}}} \equiv_{(\mathbb{N}\to\mathbb{N})\to\mathbb{P}} \lambda f.\ \forall n.\ fn = 0$, and thus $\mathsf{EPF}_{\mathbb{B}} \vee \mathsf{SCT} \to \neg\mathcal{D}(\lambda f.\ \forall n.\ fn = 0)$.

Proof. Claims (1) and (2) are easy. For (3), let $p\colon\mathbb{N}\to\mathbb{P}$ be the enumerable, undecidable predicate from Fact 7.1. Since p is semi-decidable by Corollary 4.57, we have $p \preceq_m \mathcal{K}_{\mathbb{N}\to\mathbb{B}}$ by (1) and Fact 7.2. Since $\overline{p}$ is undecidable, $\overline{\mathcal{K}}_{\mathbb{N}\to\mathbb{B}}$ is as well, and the claim follows by (2). ∎

Thus, when assuming functional extensionality, we know that $(\lambda f.\ \forall n.\ fn = 0)$ does not many-one reduce to any predicate on natural numbers.

functional extensionality
→ Sec. 7.3, Page 70

Lemma 7.4. Let $p\colon\mathbb{N}\to\mathbb{P}$ and assume functional extensionality. If $(\forall n.\ fn = 0) \preceq_m p$, then $\mathcal{D}(\forall n.\ fn = 0)$. Thus $\mathsf{EPF}_{\mathbb{B}} \vee \mathsf{SCT} \to \neg((\forall n.\ fn = 0) \preceq_m p)$.

Proof. Let $F : (\mathbb{N}\to\mathbb{N})\to\mathbb{N}$ reduce $(\forall n.\ fn = 0)$ to p.

We prove that

$$Ff = F(\lambda x.\ 0) \leftrightarrow \forall f.\ fn = 0$$

For the direction from left to right, assume $Ff = F(\lambda x.\ 0)$. We have $\forall n.\ fn = 0 \leftrightarrow p(Ff) \leftrightarrow p(F(\lambda n.\ 0)) \leftrightarrow \forall n.0 = 0$. For the direction from right to left, assume $\forall f.\ fn = 0$. By functional extensionality, we have that $f = \lambda n.\ 0$, which proves the claim

Thus, $\lambda f.\ Ff =_{\mathbb{B}} F(\lambda x.\ 0)$ decides $\forall n.\ fn = 0$. ∎

7.2 Kleene trees

[135] Kleene. 1953. Recursive functions and intuitionistic mathematics.

In a lecture in 1953 Kleene [135] gave an example how the axioms of Brouwer's intuitionism fail if all functions are considered computable by constructing an infinite decidable binary tree with no computable infinite path. The existence of such a Kleene tree (KT) is in contradiction to Brouwer's fan theorem, which we will discuss later. We prove that $\mathsf{EPF}_{\mathbb{B}}$ implies KT.

Def. binary tree

For this purpose, we call a predicate $\tau\colon\mathbb{LB}\to\mathbb{P}$ a (decidable) **binary tree** if

(a) τ is decidable: $\exists f.\forall u.\tau u \leftrightarrow f\,u = \mathsf{true}$,

(b) τ is non-empty: $\exists u.\tau u$,

(c) τ is prefix-closed: If τu_2 and $u_1 \sqsubseteq u_2$, then τu_1 (where $u_1 \sqsubseteq u_2 := \exists u'.\, u_2 = u_1 +\!\!+ u'$).

We will just speak of trees instead of decidable binary trees in the following.

Fact 7.5. For every tree τ, $\tau[\,]$ holds.

Furthermore, a decidable binary tree τ ...

- ...is **bounded** if $\exists n.\forall u.\ |u| \geq n \to \neg\tau u$.
- ...is **well-founded** if $\forall f.\exists n.\ \neg\tau[f\,0,\ldots,f\,n]$.
- ...has an **infinite path** if $\exists f.\forall n.\ \tau[f\,0,\ldots,f\,n]$.
- ...is **infinite** if $\forall n.\exists u.\ |u| \geq n \wedge \tau u$.

Def. bounded
Def. well-founded
Def. infinite path
Def. infinite tree

Fact 7.6. A tree is not bounded if and only if it is infinite.

Fact 7.7. Every bounded tree is well-founded and every tree with an infinite path is infinite.

Note that both implications are strict: In our setting we cannot prove boundedness from well-foundedness nor obtain an infinite path from infiniteness, as can be seen from a Kleene tree:

$\mathsf{KT} :=$ *There exists an infinite, well-founded, decidable binary tree.*

We follow Bauer [11] to construct a Kleene tree. First we construct a partial function d different to all total functions via diagonalisation.

[11] Bauer. 2006b. König's Lemma and Kleene Tree.

Lemma 7.8. Given $\mathsf{EPF}_{\mathbb{B}}$ one can construct $d\colon \mathbb{N} \rightharpoonup \mathbb{B}$ such that $\forall f\colon \mathbb{N}\to\mathbb{B}.\exists n b.\ dn \triangleright b \wedge fn \neq b$.

Proof. Define $dn := enn \gg\!= \lambda b.\ \mathsf{ret}\,(\neg_{\mathbb{B}} b)$. ∎

We then use d to define a Kleene tree:

$$\tau_K u := \forall n < |u|.\forall x.\ \mathsf{seval}\ (dn)\ |u| = \mathsf{Some}\,x \to u[n] = \mathsf{Some}\,x$$

Intuitively, τ_K contains all paths $u = [b_0, b_1, \ldots, b_n]$ which might be prefixes of d given n as step index, i.e. where n does not suffice to verify that d is no prefix of d. An infinite path through τ_K would be a totalisation of d, which is impossible due to Lemma 7.8.

Theorem 7.9. $\mathsf{EPF}_{\mathbb{B}} \to \mathsf{KT}$.

Proof. We prove the following:

(a) τ_K is decidable because it is defined as finite quantification over a decidable predicate.

(b) τ_K is prefix-closed because seval and thus D are stationary.

(c) $\tau_K[\,]$ is vacuously true.

(d) To show that τ_K is infinite let k be given. We define $f\,0 := [\,]$ and $f(Sn) := fn +\!\!+ [\textbf{if } Dkn \textbf{ is } \mathsf{Some}\,x \textbf{ then } x \textbf{ else } \mathsf{false}]$. We have $|fn| = n$. In particular, $|fk| \geq k$ and $\tau_K(fk)$.

(e) For well-foundedness let $f : \mathbb{N}\to\mathbb{B}$ be given. There is n such that $dn \triangleright b$ and $fn \neq b$. Thus there is k such that $\mathsf{seval}\ (dn)\ k = \mathsf{Some}\,b$. Now $\neg\tau_K u$ for $u := [f\,0,\ldots,f\,(n+k)]$.

∎

7.3 Extensionality axioms

CIC is intensional, i.e. $f \equiv_{A\to B} g$ and $f = g$ do not coincide. Extensionality properties can however be consistently assumed as axioms. In this section we briefly discuss the relationship between CT and functional extensionality Fext, propositional extensionality Pext and proof irrelevance PI, defined as follows:

$$\mathsf{Fext} :\equiv \forall AB.\forall f\, g : A\to B.\ (\forall a. f\, a = g\, a)\to f = g$$

$$\mathsf{Pext} :\equiv \forall PQ : \mathbb{P}.\ (P \leftrightarrow Q)\to P = Q$$

$$\mathsf{PI} :\equiv \forall P : \mathbb{P}.\forall (x_1 x_2 : P).\ x_1 = x_2$$

Fact 7.10. $\mathsf{Pext}\to\mathsf{PI}$

[220] Swan and Uemura. 2019. On Church's Thesis in Cubical Assemblies.

Swan and Uemura [220] prove that intensional predicative Martin-Löf type theory remains consistent if CT, the axiom of univalence, and propositional truncation are added. Since functional extensionality and propositional extensionality are a consequence of univalence, and propositions are semantically defined as exactly the irrelevant types, Fext, Pext, and PI hold in this extension of type theory. It seems likely that the consistency result can then be adapted to CIC, yielding a consistency proof for CT with Fext, Pext, and PI.

It is however crucial to formulate SCT using $\exists$ instead of Σ. Already the non-parametric formulation of SCT as

$$\mathsf{SCT}_\Sigma :\equiv \exists\phi.\ \forall f.\ \Sigma c.\forall x.\exists n.\ \phi_c^n x = \mathsf{Some}(f\, x)$$

[228] Troelstra and van Dalen. 1988. Constructivism in mathematics. Vol. I.

is inconsistent with functional extensionality Fext, as observed in [228].

Lemma 7.11. $\mathsf{SCT}_\Sigma\to\mathsf{Fext}\to\bot$

Proof. Since SCT_Σ implies SCT, it suffices to prove that $\lambda f.\forall n.\ f\, n = 0$ is decidable by Lemma 7.3. Assume $G : \forall f.\ \Sigma c.\ \forall x.\exists n.\ \phi_c^n x = \mathsf{Some}(f\, x)$ and let $F f :=$ **if** $\pi_1(Gf) = \pi_1(G(\lambda x.0))$ **then** true **else** false.

If $Ff = \mathsf{true}$, then $\pi_1(Gf) = \pi_1(G(\lambda x.0))$, and since ϕ is stationary, $f\, n = (\lambda x.0) n = 0$.

If $\forall n.\ f\, n = 0$, then $f = \lambda x.\ 0$ by Fext, thus $\pi_1(Gf) = \pi_1(G(\lambda x.\ 0))$ and $Ff = \mathsf{true}$. ∎

7.4 Classical logical axioms

In this section we consider consequences of the law of excluded middle LEM. Precisely, besides LEM, we consider the weak law of excluded middle WLEM, the Gödel-Dummett-Principle DGP,[2] and the principle of independence of premises IP, together with their respective restriction of propositions to the satisfiability of boolean functions, resulting in the limited principle of omniscience LPO, the weak limited principle of omniscience WLPO, and the lesser limited principle of omniscience LLPO.

[2] We follow Diener [55] in using the abbreviation DGP, which stands for "Dirk Gently's principle", and refer to footnotes 4 and 5 in [55] for an explanation of this joke.

$$\mathsf{LEM} := \forall P : \mathbb{P}.\ P \vee \neg P \qquad \mathsf{LPO} := \forall f : \mathbb{N} \to \mathbb{B}.\ (\exists n.\ f\,n = \mathsf{true}) \vee \neg(\exists n.\ f\,n = \mathsf{true})$$

$$\mathsf{WLEM} := \forall P : \mathbb{P}.\ \neg\neg P \vee \neg P \qquad \mathsf{WLPO} := \forall f : \mathbb{N} \to \mathbb{B}.\ \neg\neg(\exists n.\ f\,n = \mathsf{true}) \vee \neg(\exists n.\ f\,n = \mathsf{true})$$

$$\mathsf{DGP} := \forall PQ : \mathbb{P}.(P \to Q) \vee (Q \to P) \qquad \mathsf{LLPO} := \forall f\,g : \mathbb{N} \to \mathbb{B}.\ ((\exists n.\ f\,n = \mathsf{true}) \to (\exists n.\ g\,n = \mathsf{true}))$$

$$\vee\ ((\exists n.\ g\,n = \mathsf{true}) \to (\exists n.\ f\,n = \mathsf{true}))$$

$$\mathsf{IP} := \qquad \forall P : \mathbb{P}.\forall q : \mathbb{N} \to \mathbb{P}.\ (P \to \exists n.qn) \to \exists n.\ P \to qn$$

Fact 7.12. $\mathsf{LEM} \to \mathsf{DGP}$, $\mathsf{DGP} \to \mathsf{WLEM}$, and $\mathsf{LEM} \to \mathsf{IP}$.

The converses are likely not provable: Diener constructs a topological model where DGP holds but not LEM, and one where WLEM holds but not DGP [55, Proposition 8.5.3]. Pédrot and Tabareau [184] construct a syntactic model where IP holds, but LEM does not.

Fact 7.13. $\mathsf{LPO} \to \mathsf{WLPO}$ and $\mathsf{WLPO} \to \mathsf{LLPO}$.

The converses are likely not provable: Both implications are strict over IZF with dependent choice [105, Theorem 5.1].

LPO is Σ^0_1-LEM and WLPO is simultaneously Σ^0_1-WLEM and Π^0_1-LEM, due to the following:

Fact 7.14. $(\forall n.f\,n = \mathsf{false}) \leftrightarrow \neg(\exists n.f\,n = \mathsf{true})$

Both can also be formulated for predicates.

Fact 7.15. The following equivalences hold:

1. $\mathsf{LPO} \quad \leftrightarrow \forall X.\forall(p : X \to \mathbb{P}).\ \mathcal{S}p \to \forall x.\ \ px \vee \neg px$
2. $\mathsf{WLPO} \leftrightarrow \forall X.\forall(p : X \to \mathbb{P}).\ \mathcal{S}p \to \forall x.\neg px \vee \neg\neg px$
3. $\mathsf{WLPO} \leftrightarrow \forall X.\forall(p : X \to \mathbb{P}).\ \overline{\mathcal{S}}p \to \forall x.\ \ px \vee \neg px$

In our formulation, LLPO is the Gödel-Dummet rule for Σ^0_1 propositions. It can also be formulated as Σ^0_1 or $\mathcal{S}$ De Morgan rule (items (2) and (3) in the following Lemma), $\mathcal{S}$-DGP (4), or as a double negation elimination principle on $\overline{\mathcal{S}}$ relations into booleans (5).

Lemma 7.16. The following are equivalent:

1. LLPO
2. $\forall f\,g : \mathbb{N} \to \mathbb{B}.\ \neg((\exists n.f\,n = \mathsf{true}) \wedge (\exists n.gn = \mathsf{true})) \to \neg(\exists n.f\,n = \mathsf{true}) \vee \neg(\exists n.gn = \mathsf{true})$
3. $\forall X.\forall(p\ q : X \to \mathbb{P}).\ \mathcal{S}p \to \mathcal{S}q \to \forall x.\ \neg(px \wedge qx) \to \neg px \vee \neg qx$
4. $\forall X.\forall(p : X \to \mathbb{P}).\ \mathcal{S}p \to \forall xy.\ (px \to py) \vee (py \to px)$
5. $\forall X.\forall(R : X \to \mathbb{B} \to \mathbb{P}).\ \overline{\mathcal{S}}R \to \forall x.\ \neg\neg(\exists b.\ Rxb) \to \exists b.\ Rxb$
6. $\forall f.\ (\forall nm.f\,n = \mathsf{true} \to f\,m = \mathsf{true} \to n = m) \to (\forall n.f\,(2n) = \mathsf{false}) \vee (\forall n.f\,(2n+1) = \mathsf{false})$

We define the **principle of finite possibility** as $\mathsf{PFP} := \forall f.\exists g.\ (\forall n.\ f\,n = \mathsf{false}) \leftrightarrow (\exists n.\ g\,n = \mathsf{true})$. PFP unifies WLPO and LLPO.

Def. principle of finite possibility

Fact 7.17. $\mathsf{WLPO} \leftrightarrow \mathsf{LLPO} \wedge \mathsf{PFP}$

A principle unifying the classical axioms with their counterparts for Σ^0_1 is **Kripke's schema** $\mathsf{KS} := \forall P : \mathbb{P}.\exists f : \mathbb{N} \to \mathbb{B}.\ P \leftrightarrow \exists n.\ f\,n = \mathsf{true}$.

Def. Kripke's schema

Fact 7.18. LEM → KS

Fact 7.19. Given KS we have LPO → LEM, WLPO → WLEM, and LLPO → DGP.

KS could be strengthened to state that every predicate is semi-decidable (to which KS is equivalent using $\mathsf{AC}_{\mathbb{N},\mathbb{N}\to\mathbb{N}}$). The strengthening would be incompatible with CT.

In general, the compatibility of classical logical axioms (without assuming choice principles) with CT seems open. We conjecture that Coq's restriction preventing large elimination principles for non-sub-singleton propositions makes LEM and CT consistent in Coq.

7.5 Axioms of Russian constructivism

The Russian school of constructivism analyses recursive analysis based on computable functions and constructive logic under the assumption of **Markov's principle**:

Def. Markov's principle

$$\mathsf{MP} := \forall f : \mathbb{N} \to \mathbb{B}.\ \neg\neg(\exists n.\ f\,n = \mathsf{true}) \to \exists n.\ f\,n = \mathsf{true}$$

[40] Coquand and Mannaa. 2017. The Independence of Markov's Principle in Type Theory.

[184] Pédrot and Tabareau. 2018. Failure is Not an Option.

Markov's principle is independent in type theory [40, 184], consistent with CT [220], and follows from LPO.

Fact 7.20. LPO ↔ WLPO ∧ MP

Corollary 7.21. LPO → MP.

[106] Herbelin. 2010. An Intuitionistic Logic that Proves Markov's Principle.

It seems likely that the converse is not provable: There is a logic where MP holds, but not LPO [106]. As observed by Herbelin [106] and Pedrót and Tabareau [184], IP ∧ MP yields LPO.

Lemma 7.22. MP → IP → LPO

Proof. Given $f : \mathbb{N} \to \mathbb{B}$ there is $n_0 : \mathbb{N}$ s.t. $\forall k.\ f\,k = \mathsf{true} \to f\,n_0 = \mathsf{true}$ using MP and IP: By MP, $\neg\neg(\exists k.\ f\,k = \mathsf{true}) \to \exists n.\ f\,n = \mathsf{true}$ and by IP, $\exists n.\neg\neg(\exists k.f\,k = \mathsf{true}) \to f\,n = \mathsf{true}$, which suffices. Now $f\,n_0 = \mathsf{true} \leftrightarrow \exists n.\ f\,n = \mathsf{true}$ and LPO follows. ∎

Note that in the last proof, IP is used for a proposition which is neither decidable nor Σ^0_1. A nicer factorisation would be to prove IP → WLPO, but the implication seems unlikely.

Lemma 7.23. The following are equivalent:

1. MP
2. $\forall X.\forall p : X \to \mathbb{P}.\ \mathcal{S}p \to \forall x.\ \neg\neg px \to px$
3. $\forall X.\forall p : X \to \mathbb{P}.\ \mathcal{S}p \to \mathcal{S}\overline{p} \to \forall x.\ px \vee \neg px$
4. $\forall X.\forall p : X \to \mathbb{P}.\ \mathcal{S}p \to \mathcal{S}\overline{p} \to \mathcal{D}p$
5. $\forall X.\forall(R : X \to \mathbb{B} \to \mathbb{P}).\ \mathcal{S}R \to \forall x.\ \neg\neg(\exists b.\ Rxb) \to \exists b.\ Rxb$
6. $\forall p : \mathbb{N} \to \mathbb{P}.\ \mathcal{S}p \to \neg\neg(\exists x.\ px) \to \exists x.\ px$

Proof.
- (1) → (2) is immediate.
- (2) → (3): Since $\mathcal{S}$ is closed under disjunctions and since $\neg\neg(px \vee \neg px)$ is a tautology.

- (3) → (4) is immediate by Lemma 4.58 with $Rxb := (px \wedge b = \mathsf{true}) \vee (\neg px \wedge b = \mathsf{false})$.
- (4) → (1): Let $\neg\neg(\exists n.fn = \mathsf{true})$. Let $p(x : \mathbb{N}) := \exists n.fn = \mathsf{true}$. Now p is semi-decided by $\lambda x.f$, $\overline{p}$ by $\lambda xn.\mathsf{false}$, and $p0 \vee \neg p0$ by (4). One case is easy, the other contradictory. ■
- (5) → (2): Since $\exists b.Rxb$ is semi-decidable if R is semi-decidable.
- (2) → (5): Since for $Rxb := px$ we have $px \leftrightarrow \exists b.Rxb$.
- (2) ↔ (6) is straightforward.

Note that (4) is often called "Post's theorem" since it was first proved by Post [189, §1]. (1) ↔ (3) ↔ (4) is already discussed in [73]. (5) is dual to Lemma 7.16 (5). Replacing $\mathcal{S}p$ with $\overline{\mathcal{S}}p$ in (2) does however not result in an equivalent of LLPO, but turns (2) into an assumption-free fact. While in general $\mathcal{S}\overline{p} \leftrightarrow \overline{\mathcal{S}}p$ does not hold it seems possible that they can be exchanged in (3) and (4), but we are not aware of a proof.

[189] Post. 1944. Recursively enumerable sets of positive integers and their decision problems.

When proving the termination of a program, many textbooks rely to a proof by contradiction, and occasionally we will need this technique as well. It is also well-known that Markov's principle MP suffices for this restricted class of proofs by contradiction, and the full power of LEM is not needed. This also holds for our definition of partial functions, as made precise by the following extension of the previous lemma:

Fact 7.24. The following are equivalent.

1. MP
2. $\forall x : \mathsf{part}\, A.\ \neg\neg(x\downarrow) \to x\downarrow$
3. $\forall x : \mathsf{part}\, A. \forall a : A.\ \neg\neg(x \triangleright a) \to x \triangleright a$

7.6 Choice axioms

We consider the axioms of functional choice AC, unique functional choice AUC, dependent functional choice ADC, countable functional choice ACC, and the axiom of relational choice UNIF as well as the special cases of functional number-number choice $\mathsf{AC}_{\mathbb{N},\mathbb{N}}$, and functional function-number choice $\mathsf{AC}_{\mathbb{N}\to\mathbb{N},\mathbb{N}}$, which are sometimes called $\mathsf{AC}_{0,0}$ and $\mathsf{AC}_{1,0}$ in the literature. The abbreviation UNIF is supposed to remind of the axiom of uniformisation in descriptive set theory, which is a restriction of UNIF for Polish spaces.

$$\mathsf{AC}_{X,Y} := \forall R : X \to Y \to \mathbb{P}.(\forall x.\exists y.Rxy) \to \exists f : X \to Y.\forall x.\, Rx(fx)$$

$$\mathsf{AUC}_{X,Y} := \forall R : X \to Y \to \mathbb{P}.(\forall x.\exists! y.Rxy) \to \exists f : X \to Y.\forall x.\, Rx(fx)$$

$$\mathsf{ADC}_X := \forall R : X \to X \to \mathbb{P}.(\forall x.\exists x'.Rxx') \to \forall x_0.\exists f : \mathbb{N} \to X.f0 = x_0 \wedge \forall n.\, R(fn)(f(n+1)))$$

$$\mathsf{UNIF}_{X,Y} := \forall R{:}\, X \to Y \to \mathbb{P}.\ (\forall x.\exists y.\, Rxy) \to \exists R'.(\forall xy.\, R'xy \to Rxy) \wedge (\forall x.\exists! y.R'xy)$$

$$\mathsf{AC} := \forall XY : \mathbb{T}.\ \mathsf{AC}_{X,Y} \qquad \mathsf{AUC} := \forall XY.\ \mathsf{AUC}_{X,Y} \quad \mathsf{ADC} := \forall X : \mathbb{T}.\ \mathsf{ADC}_X \quad \mathsf{ACC} := \forall X : \mathbb{T}.\ \mathsf{AC}_{\mathbb{N},X}$$

$$\mathsf{UNIF} := \forall XY : \mathbb{T}.\mathsf{UNIF}_{X,Y}$$

Fact 7.25. $\mathsf{AC}_{X,X} \to \mathsf{ADC}_X$, $\mathsf{AC}_{X,Y} \to \mathsf{AUC}_{X,Y}$, $\mathsf{ADC} \to \mathsf{ACC}$, $\mathsf{ACC} \to \mathsf{AC}_{\mathbb{N},\mathbb{N}}$, and $\mathsf{AC}_{\mathbb{N}\to\mathbb{N},\mathbb{N}} \to \mathsf{AC}_{\mathbb{N},\mathbb{N}}$.

The axiom of functional choice can be factored into the axiom of functional unique choice and the axiom of relational choice.

Fact 7.26. $\mathsf{AC} \leftrightarrow \mathsf{AUC} \wedge \mathsf{UNIF}$

The following well-known fact is due to Diaconescu [54] and Myhill and Goodman [99].

Fact 7.27. $\mathsf{AC} \to \mathsf{Fext} \to \mathsf{Pext} \to \mathsf{LEM}$

Given that $\mathsf{AC}_{\mathbb{N}\to\mathbb{N},\mathbb{N}}$ turns SCT into SCT_Σ, we have:

Fact 7.28. $\mathsf{AC}_{\mathbb{N}\to\mathbb{N},\mathbb{N}} \to \mathsf{Fext} \to \mathsf{SCT} \to \bot$

We will later see that $\mathsf{LLPO} \wedge \mathsf{AC}_{\mathbb{N},\mathbb{N}}$ implies weak Kőnig's lemma, which is incompatible with KT. Already now we can prove that $\mathsf{WLPO} \wedge \mathsf{AUC}_{\mathbb{N},\mathbb{B}}$ is incompatible with $\mathsf{EPF}_{\mathbb{B}}$.

Fact 7.29. $\mathsf{AUC}_{\mathbb{N},\mathbb{B}} \to (\forall n : \mathbb{N}.\ pn \vee \neg pn) \to \mathcal{D}p$

Lemma 7.30. $\mathsf{WLPO} \to \mathsf{AUC}_{\mathbb{N},\mathbb{B}} \to \mathsf{EPF}_{\mathbb{B}} \to \forall p\colon \mathbb{N}\to\mathbb{P}.\mathcal{S}p \to \mathcal{D}\overline{p}$

Proof. Let p be enumerable. WLPO implies $\forall n.\ \neg pn \vee \neg\neg pn$. By $\mathsf{AUC}_{\mathbb{N},\mathbb{B}}$ and the last lemma $\overline{p}$ is decidable. ∎

Corollary 7.31. $\mathsf{WLPO} \to \mathsf{AUC}_{\mathbb{N},\mathbb{B}} \to \mathsf{EPF}_{\mathbb{B}} \to \bot$

[9] Barras. 2010. Sets in Coq, Coq in Sets.

CT seems to be consistent with full AC in CIC, since this is true in IZF, IZF can be used to model CIC [9]. Of course, this only holds as long as classical axioms like WLPO are absent. However, it is conceivable that CT, LEM, and UNIF are consistent in CIC, since the universe of (then classical) propositions seems to be still strictly separated from any computation, because none of the axioms can be used to define actual functions $f : \mathbb{N} \to \mathbb{N}$.

7.6.1 Provable choice principles

In contrast to predicative Martin-Löf type theory, CIC does not prove the axiom of choice, nor the axioms of dependent and countable choice. This is due to the fact that arbitrary large eliminations are not allowed. However, recall that a large elimination principle for the accessibility predicate is provable, resulting in Corollary 3.8. To make this chapter self-contained, we recall the consequences of Corollary 3.8 we already proved in Chapter 4.

Using Corollary 3.8 we were then able to prove that relations with a decider, enumerator, or semi-decider have choice functions. As a consequence, CIC allows proving the following choice principles:

Lemma 7.32. Let X be a type and Y be a discrete type. Then $\mathcal{D}$-$\mathsf{AC}_{X,\mathbb{N}}$, $\mathcal{S}$-$\mathsf{AC}_{X,\mathbb{N}}$ and $\mathcal{E}$-$\mathsf{AC}_{Y,X}$ are provable, i.e.:

1. $\forall R : X \to \mathbb{N} \to \mathbb{P}.\ \mathcal{D}R \to (\forall x.\exists n.\ Rxn) \to \exists f\ : X \to \mathbb{N}.\forall x.\ Rx(f\,x)$
2. $\forall R : X \to \mathbb{N} \to \mathbb{P}.\ \mathcal{S}R \to (\forall x.\exists n.\ Rxn) \to \exists f\ : X \to \mathbb{N}.\forall x.\ Rx(f\,x)$
3. $\forall R : Y \to X \to \mathbb{P}.\ \mathcal{E}R \to (\forall y.\exists x.\ Ryx) \to \exists f\ : Y \to X.\forall y.\ Ry(f\,y)$

Proof. (1) follows immediately from Lemma 4.6. (2) and (3) can be directly proved from (1), but also directly via 4.58, and 4.26. ∎

Furthermore, in Lemma 3.29 we proved a choice principle for relations with finite, discrete domain type.

Fact 7.33. Let X be a discrete and finite type and Y any type. We have:

$$\forall R\colon X \to Y \to \mathbb{P}.\ (\forall x.\exists y.\ Rxy) \to \exists f\colon X \to Y.\forall x.\ Rx(fx)$$

Note that in particular $\mathcal{S}$-$\mathsf{AC}_{\mathbb{N},\mathbb{B}}$ follows from $\mathcal{S}$-$\mathsf{AC}_{\mathbb{N},\mathbb{N}}$.We briefly discuss consequences of the here mentioned principles with regards to CT for oracles and in the next section $\overline{\mathcal{S}}$-$\mathsf{AC}_{\mathbb{N},\mathbb{B}}$ will be central.

7.6.2 Modesty and Oracles

Using $\mathcal{D}$-$\mathsf{AC}_{\mathbb{N},\mathbb{N}}$ from Lemma 7.32 allows proving a choice axiom w.r.t. models of computation, observed by Larchey-Wendling [150] and called "modesty" by Forster and Smolka [83].

Lemma 7.34. Let $\phi\colon \mathbb{N} \to \mathbb{N} \to \mathbb{N} \to \mathbb{ON}$ be stationary. We have

$$\forall c.(\forall x.\exists vn.\ \phi_c^n x = \mathsf{Some}\, v) \to \exists f\ :\mathbb{N} \to \mathbb{N}.\forall x.\exists n.\ \phi_c^n x = \mathsf{Some}\,(f\,x).$$

Note that this is reminiscent to the function eval on partial values defined in Lemma 4.59. We will prove the lemma concretely for a step-indexed interpreter of the weak call-by-value λ-calculus in Lemma 27.11.

That is, if c is the code of a function inside the model of computation which is provably total, the total function can be computed outside of the model. This modesty principle simplifies the mechanisation of computability theory in type theory as e.g. in [84]. For instance, it allows to prove that defining decidability as "a total function in the model of computation deciding the predicate" and as "a meta-level function deciding the predicate which is computable in the model of computation" is equivalent.

However, the modesty principle prevents a naive extension of CT to oracles. Traditionally, computability theory based on oracles is formulated using a computability function ϕ_p, such that for $p : \mathbb{N}\to\mathbb{P}$ there exists a code c_p representing a total function such that $\forall x.(\exists k.(\phi_p)_c^n x = \mathsf{Some}\,0) \leftrightarrow px$.

Synthetically, we may want to assume ϕ_p for every p as "Church's thesis with oracles". "Church's thesis with oracles" implies SCT, and we know that under SCT there is a predicate $\mathcal{K}$ which is undecidable. However, under the presence of $\mathcal{D}$-$\mathsf{AC}_{\mathbb{N},\mathbb{N}}$ we can use $\phi_{\mathcal{K}}$ and obtain $c_{\mathcal{K}}$ which can be turned into a decider $f\ :\mathbb{N}\to\mathbb{B}$ for $\mathcal{K}$ using the choice principle above – a contradiction.

We discuss a more appropriate formulation of oracles in Chapter 9, which allows working solely with SCT.

7.7 Axioms on trees

We have already introduced (decidable) binary trees and Kleene trees in Section 7.2. We now give a broader overview and give formulations of LPO, WLPO, LLPO, and MP in terms of decidable binary trees, following Berger et al. [18].

Def. subtree
Def. direct subtree

Fact 7.35. Let τ be a tree. Then $\tau_u v \coloneqq \tau(u +\!\!+ v)$ is a tree if and only if τu.

If τu holds we call τ_u a **subtree** of τ and if furthermore $u = [b]$ we call τ_u a **direct subtree** of τ.

Lemma 7.36. The following equivalences hold:

1. LPO$\leftrightarrow$ every tree is bounded or infinite.
2. WLPO$\leftrightarrow$ every tree is infinite or not infinite.
3. LLPO$\leftrightarrow$ every infinite tree has a direct infinite subtree.
4. MP$\leftrightarrow$ if a tree is not infinite it is bounded.
5. MP$\leftrightarrow$ if a tree has no infinite path it is well-founded.

Def. fan theorem

Def. weak Kőnig's lemma

Recall Fact 7.7 stating that every bounded tree is well-founded and that every tree with an infinite path is infinite. The respective converse implications are known as Brouwer's **fan theorem** FAN and **weak Kőnig's lemma** WKL respectively:

$$\mathsf{FAN} :\equiv \textit{Every well-founded decidable binary tree is bounded.}$$

$$\mathsf{WKL} :\equiv \textit{Every infinite decidable binary tree has an infinite path.}$$

Fact 7.37. $\mathsf{KT} \to \neg\mathsf{FAN}$ and $\mathsf{KT} \to \neg\mathsf{WKL}$.

Note that FAN is called FAN'_Δ in [122] and FAN_Δ in [55], and WKL is called $\mathsf{WKL}_{\mathcal{D}}$ in [75]. Ishihara [122] shows how to deduce FAN from WKL constructively:

Fact 7.38. Bounded trees τ have a longest element, i.e. $\exists u.\, \tau u \wedge \forall v.\, \tau v \to |v| \leq |u|$.

Lemma 7.39. For every tree τ there is an infinite tree τ' such that for any infinite path f of τ' $\forall u.\, \tau u \to \tau[f0, \ldots, f|u|]$.

Proof. Let τ be given and $\tau' u :\equiv \tau u \vee \exists v \sqsubseteq u.\tau v \wedge \neg\exists w.|w| = |v| + 1 \wedge \tau w$.

It's obvious that τ' is inhabited and decidable. Proving that τ' is closed under prefixes is not trivial and explained in [122, Proposition 2].

To prove that τ' is infinite, let $n : \mathbb{N}$ be given. Since τ is decidable it is also decidable whether $\exists u.\tau u \wedge |u| = n$. If there is such a u, $\tau' u$ also holds. If there is no such u, τ is bounded and by Fact 7.38 there is a longest element v in τ. Then $\tau' w$ for $w :\equiv v +\!\!+ [0, \ldots, 0]$ (with n times 0) holds and $|w| \geq n$.

Now let $\forall n.\tau'[f0, \ldots, fn]$ and τu. Since τ is decidable we can assume $\neg\tau[f0, \ldots, f|u|]$ and have to obtain a contradiction. Since $\tau'[f0, \ldots, f|u|]$ either $\tau'[f0, \ldots, f|u|]$ and the contradiction is immediate or there is $v \sqsubseteq [f0, \ldots, f|u|]$ s.t.τv and $\neg\exists w.|w| = |v| + 1 \wedge \tau w$. Thus $|v| \leq |u|$. Case distinction:

1. $|v| < |u|$. Then if $u_1 = [x_1, \ldots, x_n]$ we have $\tau[x_1, \ldots, x_{|v|+1}]$, a contradiction.
2. $|v| = |u|$. Then $v = [f0, \ldots, f|u|]$. But τv and $\neg[f0, \ldots, f|u|]$. ∎

Theorem 7.40. $\mathsf{WKL} \to \mathsf{FAN}$

Proof. By Lemma 7.39 and WKL, for every τ there is f s.t. $\forall a.\, \tau u \to \tau[f0, \ldots, f|u|]$. If τ is well-founded, there is n s.t. $\neg\tau[f0, \ldots, fn]$. Then n is a bound for τ: For u with $|u| > n$ and τu we have $\tau[f0, \ldots, fn, \ldots, f|u|]$. But then $\tau[f0, \ldots, fn]$, contradiction. ∎

Corollary 7.41. $\mathsf{KT} \to \neg\mathsf{WKL}$

Berger and Ishihara [17] show that $\mathsf{FAN} \leftrightarrow \mathsf{WKL}!$, a restriction of WKL stating that every infinite decidable binary tree with *at most one* infinite path has an infinite path. Schwichtenberg [104] gives a more direct construction and mechanises the proof in Minlog.

Berger, Ishihara, and Schuster [18] characterise WKL as the combination of the logical principle LLPO and the function existence principle $\overline{\mathcal{S}}\text{-}\mathsf{AC}_{\mathbb{N},\mathbb{B}}$ (called $\Pi^0_1\text{-}\mathsf{ACC}^\vee$ in [18]). We observe that WKL can also be characterised as one particular choice or dependent choice principle. The proofs are essentially rearrangements of [18, Theorem 27 and Corollary 5]:

Theorem 7.42. The following are equivalent:

1. WKL
2. $\mathsf{LLPO} \wedge \overline{\mathcal{S}}\text{-}\mathsf{AC}_{\mathbb{N},\mathbb{B}}$
3. $\forall R\colon\ \mathbb{N}\to\mathbb{B}\to\mathbb{P}.\ \overline{\mathcal{S}}R \to (\forall n.\neg\neg\exists b.Rnb) \to \exists f\colon\mathbb{N}\to\mathbb{B}.\forall n.\ R\ n\ (f\,n)$
4. $\forall R\colon \mathbb{L}\mathbb{B}\to\mathbb{B}\to\mathbb{P}.\ \overline{\mathcal{S}}R \to (\forall u.\neg\neg\exists b.Rub) \to \exists f\colon\mathbb{N}\to\mathbb{B}.\forall n.\ R\ [f\,0,\dots,f\,(n-1)]\ (f\,n)$

Proof. For $\mathsf{WKL}\to\mathsf{LLPO}$ we use the characterisation (3) of LLPO from Lemma 7.36. Let τ be an infinite tree. By WKL there is an infinite path f. Then $\tau_{[f\,0]}$ is a direct infinite subtree.

For $\mathsf{WKL}\to\overline{\mathcal{S}}\text{-}\mathsf{AC}_{\mathbb{N},\mathbb{B}}$ let R be total and f s.t. $\forall nb.\ Rnb \leftrightarrow \forall m.f\,nbm = \mathsf{false}$. Define the tree $\tau u := \forall i < |u|.\forall m < |u|.\ f\,i(u[i])m = \mathsf{false}$. Infinity of τ follows from $\forall n.\exists u.|u| = n \wedge \forall i < n.Ri(u[i])$, proved by induction on n using totality of R. If g is an infinite path of τ, $Rn(gn)$ follows from $\forall m.\tau[g0,\dots,g(n+m+1)]$.

(2) → (3) is immediate using characterisation (3) of LLPO from Lemma 7.16.

For (3) → (4) let $F : \mathbb{N}\to\mathbb{L}\mathbb{B}$ and $G : \mathbb{L}\mathbb{B}\to\mathbb{N}$ invert each other.[3] Let $R : \mathbb{L}\mathbb{B}\to\mathbb{B}\to\mathbb{P}$ and f be the choice function obtained from (3) for $\lambda nb.R(Fn)b$. Then $\lambda n.f(G(gn))$ where $g0 := []$ and $g(\mathsf{S}\,n) := gn +\!\!+ [f(G(gn))]$ is a choice function for R as wanted.

For (4)→(1) let τ be an infinite tree and let $d_u m := \exists v.|v| = m \wedge \tau_u v$, i.e. $d_u m$ if τ_u has depth at least m and in particular τ_u is infinite iff $\forall m.d_u m$. Define $Rub := \forall m.d_{u+\!\!+[b]}m \vee \neg d_{u+\!\!+[\neg_{\mathbb{B}} b]}$. R is co-semi-decidable (since d is decidable), and $\neg R\,u\,\mathsf{true} \wedge \neg R\,u\,\mathsf{false}$ is contradictory. Thus (4) yields a choice function f which fulfils $\tau[f\,0,\dots,f\,n]$ by induction on n. ∎

Diener [55] explains the connection of WKL with the model existence theorem occurring in the completeness proof of classical propositional logic. Forster, Kirst, and Wehr [75] analyse this connection for classical first-order logic.

[75] Forster, Kirst, and Wehr. 2021b. Completeness theorems for first-order logic analysed in constructive type theory.

7.8 Continuity: Baire space, Cantor space, and Brouwer's intuitionism

The total function space $\mathbb{N}\to\mathbb{N}$ is often called **Baire space**, whereas $\mathbb{N}\to\mathbb{B}$ is called **Cantor space**. In this chapter we write $\mathbb{N}^{\mathbb{N}}$ and $\mathbb{B}^{\mathbb{N}}$ for the spaces.

Def. Baire space

Def. Cantor space

Constructively, one cannot prove that $\mathbb{N}^{\mathbb{N}}$ and $\mathbb{B}^{\mathbb{N}}$ are in bijection. However, KT is equivalent to the existence of a continuous bijection $\mathbb{B}^{\mathbb{N}}\to\mathbb{N}^{\mathbb{N}}$ with a continuous modulus of continuity, i.e. a modulus function which is continuous (in the point) itself [55]. Furthermore, KT yields a continuous bijection $\mathbb{N}^{\mathbb{N}}\to\mathbb{B}^{\mathbb{N}}$ [15].

We call a function $F : A^{\mathbb{N}}\to B^{\mathbb{N}}$ **continuous** if $\forall f : A^{\mathbb{N}}.\forall n : \mathbb{N}.\exists L : \mathbb{L}\mathbb{N}.\forall g : A^{\mathbb{N}}.\ (\mathsf{map}\ f\ L =$

Def. continuous

[3] These so called coding functions is easy to construct even formally using e.g. techniques from [73].

Def. modulus of continuity

$\mathsf{map}\ g\ L) \to Ffn = Fgn$. A function $M : A^{\mathbb{N}} \to \mathbb{N} \to \mathbb{L}\mathbb{N}$ is called the **modulus of continuity** for F if $\forall n : \mathbb{N}. \forall f\, g : A^{\mathbb{N}}.\ \mathsf{map}\ f\ (Mfn) = \mathsf{map}\ g\ (Mfn) \to Ffn = Fgn$. We define:

$$\mathsf{Homeo}(A^{\mathbb{N}}, B^{\mathbb{N}}) := \exists F : A^{\mathbb{N}} \to B^{\mathbb{N}}. \exists M.\ M \textit{ is a continuous modulus of continuity for } F$$

Def. leaf of a Kleene tree

We start by proving that $\mathsf{KT} \leftrightarrow \mathsf{Homeo}(\mathbb{B}^{\mathbb{N}}, \mathbb{N}^{\mathbb{N}})$. To do so, we say that $u +\!\!+ [b]$ is a **leaf of a Kleene tree** τ_K if $\tau_K u$, but $\neg\tau_K(u +\!\!+ [b])$.

Fact 7.43. For every τ_K, there is an injective enumeration $\ell : \mathbb{N} \to \mathbb{L}\mathbb{B}$ of the leaves of τ_K.

We define $F(f : \mathbb{N} \to \mathbb{N})n := (\ell(f0) +\!\!+ \cdots +\!\!+ \ell(f(n+1)))[n]$. Since leaves cannot be empty, the length of the accessed list is always larger than n and F is well-defined.

Lemma 7.44. F is injective w.r.t. $\equiv_{\mathbb{N}^{\mathbb{B}}}$ and $\equiv_{\mathbb{N}^{\mathbb{N}}}$.

Lemma 7.45. F is continuous with continuous modulus of continuity.

Lemma 7.46. The following hold for a Kleene tree τ_K:

1. There is a function $\ell^{-1} : \mathbb{L}\mathbb{B} \to \mathbb{N}$ such that for all leafs l, $\ell(\ell^{-1}l) = l$.
2. For all l such that $\neg\tau_K l$ there exists $l' \sqsubseteq l$ such that l' is a leaf of τ_K.
3. There is $\mathsf{pref} : (\mathbb{N} \to \mathbb{B}) \to \mathbb{L}\mathbb{B}$ such that $\mathsf{pref}\, g$ is a leaf of τ_K and $\exists n.\ \mathsf{pref}\, g = \mathsf{map}\ g\ [0, \ldots, n]$.

We can now define the inverse as $G\ g\ n := \ell^{-1}(\mathsf{pref}\,(\mathsf{nxt}^n g))$ where $\mathsf{nxt}\ g\ n := g(n + |\mathsf{pref}\, g|)$.

Lemma 7.47. $F\ (G\ g) \equiv_{\mathbb{N}\to\mathbb{B}} g$

Lemma 7.48. G is continuous with continuous modulus of continuity.

To implement F and G in Coq the theorems on lists from the `std++` Coq library were very valuable [224].

The following proof is due to Diener [55, Proposition 5.3.2].

Lemma 7.49. $\mathsf{Homeo}(\mathbb{B}^{\mathbb{N}}, \mathbb{N}^{\mathbb{B}}) \to \mathsf{KT}$

Proof. Let F be a bijection with continuous modulus of continuity M. Then $\tau u := \forall 0 < i \leq |u|. \exists k < i. k \in M(\lambda n.\textbf{if}\ l[n]\ \textbf{is}\ \mathsf{Some}\, b\ \textbf{then}\ b\ \textbf{else}\ \mathsf{false})\ 0$ is a Kleene tree. ∎

Theorem 7.50. $\mathsf{KT} \leftrightarrow \mathsf{Homeo}(\mathbb{B}^{\mathbb{N}}, \mathbb{N}^{\mathbb{N}})$ and $\mathsf{KT} \to \mathsf{Homeo}(\mathbb{N}^{\mathbb{N}}, \mathbb{B}^{\mathbb{N}})$.

Deiser [51] proves in a classical setting that $\mathsf{Homeo}(\mathbb{N}^{\mathbb{N}}, \mathbb{B}^{\mathbb{N}})$ holds. It would be interesting to see whether the proof can be adapted to a constructive proof $\mathsf{WKL} \to \mathsf{Homeo}(\mathbb{N}^{\mathbb{N}}, \mathbb{B}^{\mathbb{N}})$.

We have already seen that CT is inconsistent with FAN. Besides FAN, in Brouwer's intuitionism the continuity of functionals $\mathbb{N}^{\mathbb{N}} \to \mathbb{N}$ is routinely assumed:

$$\mathsf{Cont} := \forall F : (\mathbb{N}\to\mathbb{N})\to\mathbb{N}.\ \forall f : \mathbb{N}\to A. \exists L : \mathbb{L}\mathbb{N}. \forall g : \mathbb{N}\to A.\ (\mathsf{map}\ f\ L = \mathsf{map}\ g\ L) \to Ff \equiv_B Fg$$

Since every computable function is continuous, we believe Cont to be consistent with CT. Combining Cont with $\mathsf{AC}_{\mathbb{N}\to\mathbb{N},\mathbb{N}}$ yields **Brouwer's continuity principle**,[4] called WC-N in [228]:

Def. Brouwer's continuity principle

$$\mathsf{WC\text{-}N} := \forall R : (\mathbb{N}\to\mathbb{N})\to\mathbb{N}\to\mathbb{P}. (\forall f. \exists n. Rfn) \to \forall f. \exists Ln. \forall g.\ \mathsf{map}\ f\ L = \mathsf{map}\ g\ L \to Rgn$$

[4] But note that $\mathsf{Cont} \to \mathsf{AC}_{\mathbb{N}\to\mathbb{N},\mathbb{N}} \to \bot$, since the resulting modulus of continuity function allows for the construction of a non-continuous function [114].

Theorem 7.51. WC-N → Cont

WC-N is inconsistent with SCT, since the relation between codes and functions is not continuous:

Theorem 7.52. WC-N → SCT → ⊥

Proof. Recall that if two functions have the same code they are extensionally equal. By SCT, $\lambda f\, c.\forall x.\exists n.\phi_c^n x = \mathsf{Some}(f\,x)$ is a total relation. Using WC-N for this relation and $\lambda x.\ 0$ yields a list L and a code c s.t. $\forall g.\ \mathsf{map}\ g\ L = [0,\dots,0] \to \forall x.\exists n.\phi_c^n x = \mathsf{Some}(g\,x)$.

The functions $\lambda x.\ 0$ and $\lambda x.$ **if** $x \in L$ **then** 0 **else** 1 both fulfil the hypothesis and thus have the same code – a contradiction since they are not extensionally equal. ■

7.9 CIC as basis for constructive reverse mathematics

In this chapter we surveyed the known connections of axioms in CIC, a constructive type theory with a separate, impredicative universe of propositions, with a special focus on Church's thesis CT and formulations of axioms in terms of notions of synthetic computability.

In constructive mathematics, countable choice is often silently assumed, as critised e.g. by Richman [199, 200]. In contrast, constructive type theory with a universe of propositions seems to be a suitable base system for matters of constructive (reverse) mathematics sensitive to applications of countable choice. Due to the separate universe of propositions, such a constructive type theory neither proves countable nor dependent choice, allowing equivalences like the one in Theorem 7.42 to be stated sensitively to choice. We conjecture that Lemma 7.32 deducing $\mathcal{S}$-$\mathsf{AC}_{X,\mathbb{N}}$ and $\mathcal{E}$-$\mathsf{AC}_{\mathbb{N},X}$ directly from $\mathcal{D}$-$\mathsf{AC}_{X,\mathbb{N}}$ cannot be significantly strengthened. The proof of $\mathcal{D}$-$\mathsf{AC}_{X,\mathbb{N}}$ in turn crucially relies on a large elimination principle for $\exists n.\ f\,n = \mathsf{true}$ (Corollary 3.8). The theory of [18] proves $\mathcal{D}$-$\mathsf{AC}_{\mathbb{N},\mathbb{B}}$ and thus likely also $\mathcal{S}$-$\mathsf{AC}_{\mathbb{N},\mathbb{B}}$.

[199] Richman. 2000. The fundamental theorem of algebra: a constructive development without choice.

[200] Richman. 2001. Constructive Mathematics without Choice.

Based on the current state of knowledge in the literature it seems likely that $\mathcal{S}$-$\mathsf{AC}_{\mathbb{N},\mathbb{B}}$ and LEM together do not suffice to disprove CT, which seems to require at least classical logic of the strength of LLPO and a choice axiom for *co*-semi-decidable predicates. Thus we conjecture that a consistency proof of e.g. LEM ∧ CT might be possible for CIC. We discuss consistency and admissibility of CT for CIC in Section 29.3.

In predicative Martin-Löf type theory (MLTT), there is no universe of propositions. If one defines ∃ := Σ, MLTT proves AC. Thus, it is less versatile than CIC as basis for constructive reverse mathematics. If one wants to develop synthetic computability theory, the assumption of SCT makes the theory anti-classical: Not even LLPO can be assumed. If one defines ∃ := ¬¬Σ, MLTT does not prove AC, but MP. Consequently, one would have to define ∨ in terms of + with double negation, leading to LEM being provable as well. Thus, again, it is less versatile than CIC as basis for constructive reverse mathematics.

Type theories with propositional truncation and a semantic notion of (homotopy) propositions (such as univalent type theories) prove $\mathsf{AUC}_{\mathbb{N},\mathbb{B}}$. If one wants to develop synthetic computability theory, the assumption of SCT makes the theory anti-classical: WLPO cannot be assumed.

Lastly, the Lean proof assistant implements a version of CIC [49]. Most parts of its standard library assumes propositional extensionality Pext, a quotient type axiom implying Fext, and a choice operator [6]. The initial development of μ-recursive functions in Lean [26] was transitively based on Fext and the choice operator since it is used in standard number-theoretic facts for $\mathbb{N}$, meaning CT could not be consistently assumed. However, recently the standard

[49] de Moura, Kong, Avigad, Van Doorn, and von Raumer. 2015. The Lean theorem prover (system description).

[6] Avigad, de Moura, and Kong. 2015. Theorem proving in Lean.

[26] Carneiro. 2019. Formalizing Computability Theory via Partial Recursive Functions.

library was re-factored such that μ-recursive functions do not depend on classical axioms, and consequently CT could be assumed.

Moreover, results using the choice operator however have to be explicitly prefixed with the `noncomputable` keyword. Thus, developing synthetic computability theory seems to be feasible as long as the `noncomputable` keyword is forbidden, however comes at the price of not being able to use big parts of the standard library.

CHAPTER 8

Reducibility Degrees

In this chapter, we develop the theory of reducibility degrees by analysing the order-structure of many-one, one-one, and truth-table reducibility introduced in Chapter 5, roughly to the extent as it is covered by Post [189]. We have already proved that $\preceq_1$, $\preceq_m$, and $\preceq_{\mathrm{tt}}$ form upper semi-lattices and how to characterise $\preceq_m$ and $\preceq_{\mathrm{tt}}$ in terms of $\preceq_1$, results that did not require a universal function and thus hold without the assumption of any axioms.

[189] Post. 1944. Recursively enumerable sets of positive integers and their decision problems.

By assuming the enumerability axiom EA introduced in Chapter 6, i.e. a parametrically universal enumerator, we can prove textbook results relying on universal machines. In his seminal paper, Post was driven by the question whether the class of enumerable but undecidable predicates is equivalent (under Turing reducibility) to the halting problem, or whether there exist enumerable but undecidable problems strictly between the decidable problems and the halting problem. This question became known as Post's problem, but Post was unable to answer the question himself and instead introduced one-one, many-one, and truth-table reducibility, and posed and solved Post's problem for these notions by introducing simple and hypersimple predicates. We here follow the same route and give synthetic answers to Post's problem for $\preceq_m$ and $\preceq_{\mathrm{tt}}$, as corollaries yielding that $\preceq_1$, $\preceq_m$, and $\preceq_{\mathrm{tt}}$ are all distinct.

The central results in this chapter are

1. Post's problem for $\preceq_m$: Construction of a simple predicate, which is enumerable, undecidable, and m-incomplete.
2. Post's problem for $\preceq_{\mathrm{tt}}$: Construction of a hypersimple predicate, which is enumerable, undecidable, and tt-incomplete.

Textbook proofs of reducibility theory are heavily classical, but we manage to fully constructivise all of the results in the present chapter. The successful constructivisation mostly relies on suitable constructively weak or strong definition of textbook notions.

For Post's problem for $\preceq_m$, the definition of a simple predicate in the synthetic setting is most interesting. The complement of a simple predicate has to be infinite, but is not allowed to have an infinite, enumerable subpredicate. In classical mathematics, p is infinite if and only if it is Cantor-infinite, i.e. if there is an injective function $\mathbb{N} \hookrightarrow p$. However, Cantor-infinite predicates (defined via functions) have a (synthetically) enumerable, infinite subpredicate – enumerated by the function witnessing Cantor infinity. In constructive mathematics, a predicate p is called infinite if for any sequence $[x_1, \dots, x_n]$ there exists a y different from all x_i but such that $p\,y$. However, any fully constructive proof of infinity in this sense can be turned into a proof of Cantor infinity, meaning there can be no such proof for the complement of a simple predicate. It is thus crucial that infinity is defined to be exactly non-finiteness. The complement of a simple predicate is then non-finite, but not Cantor-infinite. Only with this definition of infinity Post's problem for $\preceq_m$ can be settled constructively.

For Post's problem for $\preceq_{\mathrm{tt}}$ several interesting aspects of constructivisation appear: First, the definition of hypersimple predicates has to be chosen carefully to ensure that hypersimple predicates are constructively tt-incomplete. The construction of a hypersimple predicate H as

the deficiency predicate of the halting problem is then easier. The literature contains multiple undecidability proofs of H, and we discuss three of them: The most conventional proof is to prove undecidability via simpleness of H. But since proofs showing that hypersimple predicates are simple seem to be inherently classical and we only manage to weaken the assumption to MP we also give a direct, fully constructive undecidability proof for H, which however does not generalise to arbitrary hypersimple predicates. Lastly, in the next chapter, we propose a definition of Turing reducibility following Bauer and show that the halting problem Turing reduces to H, again using MP.

The proofs in this chapter require the most involved intuitions of the thesis. We thus try to give intuitive conceptual outlines, but focus on the interesting synthetic and constructive aspects more than the proof ideas. Since we largely follow the excellent book by Rogers [202], supplemented by the books by Cutland [44], Soare [210], and Odifreddi [180], it might be helpful for non-experts in computability theory to consult one of the books in cases where the presented intuition is not sufficient.

[202] Rogers. 1987. Theory of Recursive Functions and Effective Computability.

[44] Cutland. 1980. Computability.

[210] Soare. 1999. Recursively enumerable sets and degrees: A study of computable functions and computably generated sets.

[180] Odifreddi. 1992. Classical recursion theory: The theory of functions and sets of natural numbers.

Outline We recap the basics of synthetic computability theory based on the axiom EA in Section 8.1. Simple predicates solving Post's problem for $\preceq_m$ are introduced in Section 8.3 and constructed in Section 8.4. We construct a truth-table complete simple predicate S^* in Section 8.5 and formalise the solution of Post's problem for $\preceq_{\mathrm{tt}}$ by introducing and constructing hypersimple predicates in Sections 8.6 and 8.7.

Publications This chapter contains adapted pieces of text from the following publication, which were written solely by the author of this thesis.

[72] Forster, Jahn, and Smolka. "A Constructive and Synthetic Theory of Reducibility: Myhill's Isomorphism Theorem and Post's Problem for Many-one and Truth-table Reducibility in Coq." *Pre-print*.

8.1 An m-complete predicate

To develop synthetic computability theory agnostic towards classical axioms like LEM, we assume the enumerability axiom EA. The axiom EA postulates

EA → Sec. 6.4, Page 59

Def. universal enumerator

1. a **universal enumerator** $\varphi : \mathbb{N} \to (\mathbb{N} \to \mathbb{ON})$
2. such that for all $p : \mathbb{N} \to \mathbb{N} \to \mathbb{P}$

$$(\exists f. \forall i.\ f_i \textit{ enumerates } p_i) \to \exists \gamma : \mathbb{N} \to \mathbb{N}. \forall i.\ \varphi_{\gamma i} \textit{ enumerates } p_i$$

To ease language, we often refer to (2) as EA in this chapter.
EA implies that φ in particular is an enumeration of all enumerable predicates.

Fact 8.1. $\forall p : \mathbb{N} \to \mathbb{P}.\ \mathcal{E} p \to \exists c.\ \varphi_c \textit{ enumerates } p$

Sometimes the following variant of EA concerned with enumerability rather than parametric enumerability is easier to use,

Lemma 8.2. Let X be enumerable and discrete and $p : X \to \mathbb{N} \to \mathbb{P}$ be enumerable. Then $\exists \gamma : \mathbb{N} \to \mathbb{N}. \forall x.\ \varphi_{\gamma x} \textit{ enumerates } px$.

For many applications it is more convenient to work with a general parameter type I.

Lemma 8.3. Let I be a discrete, enumerable type and $p: I \to \mathbb{N} \to \mathbb{P}$. We have

$$(\exists f.\forall i.\ f_i \text{ enumerates } p_i) \to \exists \gamma: \mathbb{N} \to \mathbb{N}.\forall i.\ \varphi_{\gamma i} \text{ enumerates } p_i$$

As is common in developments of computability, we start by defining an enumerable, undecidable, m-complete predicate:

$$\mathcal{W}_c x := \exists n.\ \varphi_c n = \mathsf{Some}\, x$$

We call $\mathcal{W}$ the **universal table** of enumerable predicates, justified by the following property: **Def.** universal table

Fact 8.4. $\forall p: \mathbb{N} \to \mathbb{P}.\ \mathcal{E}p \leftrightarrow \exists c.\ \forall x.\ \mathcal{W}_c x \leftrightarrow px$

The universal table $\mathcal{W}$ is enumerable, undecidable and m-complete, i.e. takes the role of the halting problem from textbook computability. We define the predicate $\mathcal{K}: \mathbb{N} \to \mathbb{P}$ as the diagonal of $\mathcal{W}$. $\mathcal{K}$ will play a similar role as the self-halting problem, instead of the codes halting on themselves, $\mathcal{K}c$ holds if c is in the range of φ_c.

m-complete
→ Sec. 5.1, Page 47

$$\mathcal{K}c := \mathcal{W}_c c$$

As in textbook proofs that $\mathcal{K}$ and $\mathcal{W}$ are undecidable we start by showing that the complement of $\mathcal{K}$ is not enumerable. Thus $\mathcal{K}$ is undecidable, and undecidability can be transported to $\mathcal{W}$ via a many-one reduction.

Lemma 8.5. $\neg\mathcal{E}\overline{\mathcal{K}}$

Proof. If $\mathcal{E}\overline{\mathcal{K}}$ there is c s.t. $\forall x.\ \mathcal{W}_c x \leftrightarrow \overline{\mathcal{K}}x \leftrightarrow \neg\mathcal{W}_c x$ by Fact 8.1. In particular $\mathcal{W}_c c \leftrightarrow \neg\mathcal{W}_c c$. Contradiction. ∎

Lemma 8.6. $\mathcal{K} \preceq_m \mathcal{W}$

Proof. By $f\,c := (c, c)$. ∎

Corollary 8.7. $\neg\mathcal{D}\mathcal{K}$, $\neg\mathcal{D}\overline{\mathcal{K}}$, $\neg\mathcal{E}\overline{\mathcal{W}}$, $\neg\mathcal{D}\overline{\mathcal{W}}$, $\neg\mathcal{D}\mathcal{W}$.

To show the enumerability of both $\mathcal{K}$ and $\mathcal{W}$, we show the enumerability of $\mathcal{W}$ and again transport via the above many-one reduction, this time positively.

Lemma 8.8. $\mathcal{E}\mathcal{W}$

Proof. By $\lambda\langle n, m\rangle$.**if** $\varphi_n m$ **is** $\mathsf{Some}\, k$ **then** $\mathsf{Some}\,(n, k)$ **else** None. ∎

Corollary 8.9. $\mathcal{E}\mathcal{K}$

We now turn towards m-completeness of $\mathcal{W}$, i.e. that all $p: X \to \mathbb{P}$ for enumerable, discrete X many-one reduce to $\mathcal{W}$:

Lemma 8.10. $\mathcal{W}$ is m-complete.

Proof. Let p be enumerable by φ_c via Fact 8.1 Then $\lambda x.(c, x)$ reduces p to $\mathcal{W}$. ∎

Establishing that $\mathcal{K}$ is m-complete as well now for the first time requires the full strength of EA, whereas before the non-parametric Fact 8.1 would have sufficed.

Lemma 8.11. $\mathcal{W} \preceq_m \mathcal{K}$

Proof. We obtain the reduction function γ from Lemma 8.3 with $p(x,y)z := \mathcal{W}_x y$. Since $\forall xyz.\, \mathcal{W}_{\gamma(x,y)}z \leftrightarrow \mathcal{W}xy$ we have $\mathcal{W}_x y \leftrightarrow \mathcal{W}_{\gamma(x,y)}(\gamma(x,y)) \leftrightarrow \mathcal{K}(\gamma(x,y))$. ∎

Corollary 8.12. $\mathcal{W} \equiv_m \mathcal{K}$ and $\mathcal{K}$ is m-complete.

8.2 Enumerable, infinite predicates

In the next section we will define simple predicates. The complement of simple predicates is infinite and does not contain an infinite, enumerable sub-predicate. In our setting of synthetic computability, the formalisation of infinity has to be carefully chosen. In Section 3.5 we have discussed three notions of infinity: Non-finiteness, generativity, and Cantor-infinity.

We will see here that only non-finiteness can be used in the definition of simple predicates, since otherwise the existence of hypersimple predicates is either disprovable or logically independent.

We start by a general discussion of the three notions in connection with enumerability. Recall that we have proved that non-finite predicates are generative if and only if LEM holds in Corollary 3.42. However, semi-decidable, non-finite predicates on natural numbers are generative already if MP holds:

MP → Sec. 3.1, Page 20

Lemma 8.13. Assume MP and let $p : \mathbb{N} \to \mathbb{P}$. Then $\mathcal{S}p \to \neg\mathcal{F}p \to \mathcal{G}p$.

Proof. Let p be semi-decidable and non-finite, and let l be given. We have to prove $\exists x.\, x \notin l \wedge px$.

Since p is semi-decidable, so is $\lambda x.\, x \notin l \wedge px$. Using characterisation (6) of MP from Lemma 7.23 it suffices to prove $\neg\neg\exists x.\, x \notin l \wedge px$, which holds by Lemma 3.33. ∎

In general, using injective countability as formalisation of infinity is perfectly valid in constructive mathematics. However, when seeing constructive type theory from the perspective of synthetic computability theory, injectively enumerable predicates contain computable structure, which usually is not implied by infinity: The injective function $f : \mathbb{N} \to X$ can be obtained from an injective enumerator but also can be turned into an injective enumerator of its range.

We start by the former and show that generative, enumerable predicates are Cantor infinite:

Lemma 8.14. Generative enumerable predicates over discrete types have a strong injective enumerator: Let X be a type and $p : X \to \mathbb{P}$. Then $\mathcal{G}p \to \mathcal{E}p \to \exists f.\, f \text{ injective} \wedge \forall x.\, px \leftrightarrow \exists n.\, fn = x$.

Proof. Let f enumerate p. We construct a function $\mathsf{nxt} : \mathbb{L}X \to X$ s.t. $\forall l.\, \mathsf{nxt}\, l \notin l \wedge p(\mathsf{nxt}\, l) \wedge \forall n_1 n_2 x.\, fn_1 = \mathsf{Some}(\mathsf{nxt}\, l) \to fn_2 = \mathsf{Some}\, x \to x \notin l \to n_1 \leq n_2$ using $\mu_{\mathbb{N}}$ from Corollary 3.8 and generativity of p.

Then for $g0 := []$, $g(\mathsf{S}\, n) := gn +\!\!+ [\mathsf{nxt}\, (gn)]$ and $Fn := \mathsf{nxt}\, (gn)$ we have

1. $n < m \to Fn \in gm$
2. $Fn \notin gn$
3. $p(Fn)$
4. $fn = \mathsf{Some}\, x \to x \in g(\mathsf{S}\, n)$

Injectivity of f follows from (1) and (2), that F is a strong enumerator from (3) and (4). ∎

Corollary 8.15. Generative enumerable predicates over discrete types are Cantor-infinite.

In synthetic computability, Cantor-infinite predicates are problematic because the function $f : \mathbb{N} \to X$ can be turned into an enumerator. From $\mathbb{N} \hookrightarrow p$ we cannot conclude that p is enumerable, but that p has a Cantor-infinite, enumerable subpredicate:

Lemma 8.16. $\mathbb{N} \hookrightarrow p \to \exists q.\, \mathcal{E}q \wedge (\forall x.\, qx \to px) \wedge \mathbb{N} \hookrightarrow q$

Proof. Given p and an injection f witnessing $\mathbb{N} \hookrightarrow p$, define $qx := \exists n.\, fn = x$. Clearly, $\forall x.\, qx \to px$ since $\forall n.\, p(fn)$, and $\lambda n.\, \mathsf{Some}(fn)$ enumerates p. f still proves $\mathbb{N} \hookrightarrow q$. ∎

Since the complement of simple predicates is supposed to be infinite and not contain an infinite, enumerable sub-predicate, taking Cantor-infinity as notion of infinity is not possible for the definition of simple predicates. Since any fully constructive proof of generativity could be turned into a proof of Cantor-infinity by Lemma 3.45, we have that under the assumption of a universal enumerator φ, defining infinity as...

1. Cantor infinity proves there is no simple predicate.
2. generativity makes the existence of simple predicates logically independent.
3. non-finiteness allows to construct a simple predicate.

8.3 Simple predicates

We now turn to Post's problem for $\preceq_m$. Post's problem for $\preceq_m$ can be seen as a statement on the structure of enumerable predicates under $\preceq_m$: It shows the existence of an enumerable, m-incomplete predicate S, i.e. a predicate which is neither decidable nor equivalent to $\mathcal{W}$. Post's problem can also be seen as a statement on undecidability proofs: There are predicates S which are enumerable and undecidable, but where the undecidability proof can not be carried out by a reduction $\mathcal{W} \preceq_m S$.

[189] Post. 1944. Recursively enumerable sets of positive integers and their decision problems.

Post [189] observed that the complement of $\mathcal{K}$ is *productive*, and that productive predicates have a non-finite, enumerable subpredicate. Since productivity transports along $\preceq_m$ this means in particular that the complement of an m-complete predicate has a non-finite, enumerable subpredicate.

Thus it suffices to construct an enumerable, undecidable predicate S such that $\overline{S}$ has *no* non-finite, enumerable subpredicate. Post called such predicates **simple**, likely due to the fact that they are less hard to solve than $\mathcal{W}$. Formally, a predicate $p : \mathbb{N} \to \mathbb{P}$ is **productive** if its non-enumerability is witnessed by a function f such that for every subpredicate q of p enumerated by φ_c, p and q differ on the element fc

Def. simple

Def. productive

$$\mathsf{productive}\, p := \exists f : \mathbb{N} \to \mathbb{N}.\, \forall c.\, (\forall x.\, \mathcal{W}_c x \to px) \to p(fc) \wedge \neg \mathcal{W}_c(fc)$$

Def. simple predicate

A predicate $p : X \to \mathbb{P}$ is **simple** if it is enumerable, and its complement is both non-finite (otherwise p would at least classically be decidable) and does *not* contain a non-finite, enumerable predicate:

$$\mathsf{simple}\, p := \mathcal{E}p \wedge \neg \mathcal{F}\overline{p} \wedge \neg \exists q.\, (\forall x.\, qx \to \overline{p}x) \wedge \mathcal{E}q \wedge \neg \mathcal{F}q$$

Lemma 8.17. Productive predicates are not enumerable.

Proof. Let p have a productive function f and be enumerable by φ_c via Fact 8.1. Then by the specification of f, $p(f\,c)$, i.e. $\mathcal{W}_c(f\,c)$, and $\neg\mathcal{W}_c(f\,c)$. Contradiction. ∎

Lemma 8.18. $\overline{\mathcal{K}}$ is productive.

Proof. Pick $\lambda n.n$ as function. Let c be given s.t. $\forall x.\ \mathcal{W}_c x \to \neg\mathcal{W}xx$. In particular, $\mathcal{W}_c c \to \neg\mathcal{W}_c c$, i.e. $\neg\mathcal{W}_c c$ as required. ∎

Every productive predicate is Cantor-infinite and thus has an enumerable, Cantor-infinite subpredicate.

Lemma 8.19. Every productive predicate is Cantor-infinite.

Proof. From Lemma 8.2 and since $x \in l$ is decidable, we obtain $c : \mathbb{LN} \to \mathbb{N}$ s.t. $\mathcal{W}_{cl} x \leftrightarrow x \in l$ for every $l : \mathbb{LN}$.

Let $p : \mathbb{N} \to \mathbb{P}$ have a productive function f. We prove $\forall l : \mathbb{LN}.\Sigma x.\ (\forall x_0 \in l.px_0) \to px \wedge x \notin l$, which suffices by a slight adaption of Lemma 3.45.

Given $l : \mathbb{LN}$, pick $x := f(cl)$. ∎

Corollary 8.20. Every productive predicate has an enumerable, Cantor-infinite subpredicate.

Proof. Using Lemma 8.16. ∎

We now extend this result and show that the complement of m-complete predicates contains an enumerable, Cantor-infinite subpredicate since productiveness transports along many-one reductions:

Lemma 8.21. Let $p \preceq_m q$. If p is productive, q is productive.

Proof. Let f many-one reduce p to q, and let g be a productive function for p. Lemma 8.2 yields k s.t. $\mathcal{W}_{(kc)} x \leftrightarrow \mathcal{W}_c(f\,x)$.

Then $\lambda c.\ f\,(g(kc))$ is a productive function for q. ∎

Note that for Lemma 8.21 *parametric* universality of φ is crucial.

Lemma 8.22. Let p be m-complete. Then $\overline{p}$ has a Cantor-infinite, enumerable subpredicate.

Proof. Since productiveness of $\overline{p}$ follows from $\overline{\mathcal{K}} \preceq_m \overline{p}$ and productiveness of $\overline{\mathcal{K}}$. ∎

We conclude by showing that simple predicates are undecidable and m-incomplete:

Fact 8.23. Complements of simple predicates are not enumerable.

Corollary 8.24. Simple predicates are undecidable.

Theorem 8.25. Simple predicates are m-incomplete.

Proof. Let p be simple and m-complete. Then $\overline{p}$ contains an enumerable, Cantor-infinite and thus non-finite subpredicate. Thus p is not simple. Contradiction. ∎

We can also use simple predicates to show that $\preceq_1$ and $\preceq_m$ do not agree on an enumerable, undecidable predicate:

Lemma 8.26. If $p : \mathbb{N} \to \mathbb{P}$ and $p \times \mathbb{N} \preceq_1 p$, then p is not simple.

Proof. Let f be a one-one reduction from $p \times \mathbb{N}$ to p and let p be simple. We have to prove falsity, thus we can assume an element x_0 s.t. $\overline{p}x_0$ since $\overline{p}$ is non-finite by Fact 3.31.

Now $\lambda x.\exists n.f(x_0, n) = x$ (i.e. the range of f on $\{x_0\} \times \mathbb{N}$) is a non-finite, enumerable subpredicate of $\overline{p}$. ∎

Since for every simple predicate S, trivially $S \times \mathbb{N} \preceq_m S$ but not $p \times \mathbb{N} \preceq_1 p$ we have that $\preceq_m$ and $\preceq_1$ differ, we state the fact formally after establishing a concrete simple predicate.

8.4 Post's simple predicate

We follow the presentation of Rogers [202, §8.1 Th. II] to construct the same simple predicate as Post [189, §5].

Recall that simple predicates are enumerable and that their complement is non-finite but may not contain a non-finite, enumerable subpredicate. The latter two properties will drive the construction, since either one is easy to establish on their own, but the combination needs care. Post's idea was to construct a predicate S containing an element from every non-finite (enumerable) predicate $\mathcal{W}_c$. Thus, $\overline{S}$ cannot have a non-finite, enumerable subpredicate. To ensure that $\overline{S}$ is still non-finite, S contains only a *unique* $x > 2c$ with $\mathcal{W}_c x$ for every large enough $\mathcal{W}_c$. The condition $x > 2c$ ensures that there are at least n elements less or equal $2n$ in $\overline{S}$, and thus $\overline{S}$ is non-finite.

The only technical difficulty in the definition of S is to obtain a unique x satisfying $x > 2c$ and $\mathcal{W}_c x$ for every large enough $\mathcal{W}_c$. Post ensures this by choosing the x enumerated first by φ_c satisfying $x > 2c$, i.e. the x with the least index n such that $\varphi_c n = \mathsf{Some}\, x$. We abstract away from this property, and observe that any function mapping c to x does the job.

We fix a function $\psi : \forall c.\ (\exists x.\ \mathcal{W}_c x \wedge x > 2c) \to \mathbb{N}$ such that $\psi c H = x \to \mathcal{W}_c x \wedge x > 2c$ and $\psi c H_1 = \psi c H_2$, i.e. a proof-irrelevant choice function for the predicate $\lambda x.\ \mathcal{W}_c x \wedge x > 2c$.

We then define

$$Sx := \exists c(H : \exists x.\ \mathcal{W}_c x \wedge x > 2c).\ \psi c H = x$$

We verify S to be indeed a simple predicate:

Lemma 8.27. *S is enumerable.*

Proof. There is a strong enumerator $E : \mathbb{N} \to \mathbb{N}$ for $\lambda c.\exists x.\mathcal{W}_c x \wedge x > 2c$, i.e. we have $H\colon \forall n.\exists x.\mathcal{W}_{(En)} x \wedge x > 2 \cdot En$. Then, $\lambda n.\psi(En)(Hn)$ strongly enumerates S. ∎

Lemma 8.28. *$\overline{S}$ is non-finite.*

Proof. We prove the following, which is sufficient by Fact 3.37:

$$\forall n.\neg\neg\exists L.|L| = n \wedge \#L \wedge \forall x \in L.\overline{S}x,$$

Given n, let $px := Sx \wedge x \leq 2n$. p is exhausted by $[0, \ldots, 2n]$, thus it is not not listable. Since the claim is negative, we can assume some duplicate-free L_p listing p.

Now $|L_p| \leq n$: by decomposing Sx for every $x \in L_p$, we obtain L'_p with $\#L'_p$, $|L'_p| = |L_p|$, and $\forall c \in L'_p.\ c < n \wedge \exists H.\ \psi c H \in L_p$. Hence, $|L_p| \leq n$. Now the first n elements of $\mathsf{filter}(\lambda x.\ x \notin L_p)[0, \ldots, 2n]$ form the wanted list. ∎

Lemma 8.29. *$\overline{S}$ contains no non-finite enumerable subpredicate.*

Proof. Let q be non-finite, contained in $\overline{S}$ and enumerated by φ_c via Fact 8.1, i.e. $(*)$: $\forall x.\mathcal{W}_c x \leftrightarrow qx$. We derive a contradiction by showing $[0,\dots,2c]$ to exhaust q: Assume qx. Then $\mathcal{W}_c x$ since φ_c enumerates q. We have to prove $x \in [0,\dots,2c]$, which is decidable and thus stable. So let $x \notin [0,\dots,2c]$, i.e. $x > 2c$, and derive a contradiction. Thus we have a proof $H : \exists x.\, W_c x \land x > 2c$, and both $\mathcal{W}_c(\psi cH)$ and $S(\psi cH)$. By $(*)$ we have $q(\psi cH)$, and by assumption thus $\neg S(\psi cH)$ – contradiction. ∎

Theorem 8.30. S is a simple predicate.

Proof. Direct by Lemmas 8.27, 8.28, and 8.29. ∎

The construction of S settles Post's problem for $\preceq_m$.

Theorem 8.31. There exists an enumerable, undecidable, and m-incomplete predicate.

Proof. A function ψ can be constructed from an enumerator of $\lambda x.\mathcal{W}_c x \land x > 2c$ for every c via Lemma 4.26. The claim then follows directly from Theorem 8.25 and Theorem 8.30. ∎

Theorem 8.32. $\preceq_1$ and $\preceq_m$ differ on enumerable undecidable predicates.

Proof. $S \times \mathbb{N} \preceq_m S$ holds, but $S \times \mathbb{N} \not\preceq_1 S$ by Lemma 8.26. ∎

8.5 A tt-complete simple predicate

Post's problem for the more general notion of truth-table reducibility raises the question whether simple predicates also serve as a solution for this problem, i.e. whether they are also tt-incomplete. Post showed this is not the case in general by constructing a tt-complete simple predicate S^* extending S:

Theorem 8.33. There is a simple tt-complete predicate S^*.

Proof. We refer to [72] Appendix B and [124] for the construction of S^*. ∎

[72] Forster, Jahn, and Smolka. 2021a. A Constructive and Synthetic Theory of Reducibility: Myhill's Isomorphism Theorem and Post's Problem for Many-one and Truth-table Reducibility in Coq.

[124] Jahn. 2020. Synthetic One-One, Many-One, and Truth-Table Reducibility in Coq.

Thus, simple predicates do not solve Post's problem for truth-table reducibility, but S^* yields a distinction of $\preceq_m$ and $\preceq_{\mathrm{tt}}$ on enumerable but undecidable predicates.

Theorem 8.34. m- and tt-completeness do not coincide. In particular, $\preceq_m$ and $\preceq_{\mathrm{tt}}$ differ on enumerable undecidable predicates.

Proof. S^* is m-incomplete as a simple predicate, but tt-complete by Theorem 8.33. ∎

8.6 Hypersimple predicates

For settling Post's problem w.r.t. $\preceq_{\mathrm{tt}}$ in our synthetic setting we once more follow Rogers [202, §9.5] and introduce majorising functions: $f : \mathbb{N} \to \mathbb{N}$ **majorises** a predicate $p : \mathbb{N} \to \mathbb{P}$ if

Def. majorising function

$$\forall n.\neg\neg\exists l.\ \#l \land |l| = n \land \forall m \in l.pm \land m \leq f\,n.$$

In this section we will be slightly more liberal in inserting double negations than before, but as before with the sole goal of obtaining constructive results. In this spirit we slightly adapt the definition of majorising in order to be more suitable for constructive, formalised proofs. We immediately introduce a strengthening of the notion following Odifreddi [180]: A function $f : \mathbb{N} \to \mathbb{N}$ **exceeds** a predicate $p : \mathbb{N} \to \mathbb{P}$ if $\forall n.\neg\neg\exists i.\ n < i \leq f\,n \land pi$.

Def. exceeds

Fact 8.35. If f exceeds p, then $\lambda n.f^n 0$ majorises p.

Lemma 8.36. If $\forall x.qx \to px$ and q is Cantor-infinite, then there exists f exceeding p.

Proof. Take $\lambda n.\ \pi_1(F[0,\dots,n])$, where $F\colon \mathbb{L}\mathbb{N} \to \mathbb{N}$ is obtained from $\mathbb{N} \hookrightarrow q$ and Lemma 3.45. ∎

Corollary 8.37. Given MP, if there is no f majorising p, then $\overline{p}$ does not have a non-finite, enumerable subpredicate.

Proof. Let no f majorise p and let q be a non-finite, enumerable subpredicate of p. By MP and Lemma 8.13, q is generative. By Corollary 8.15, q is Cantor-infinite. By Lemma 8.36 there is f exceeding p. By Fact 8.35, p is majorised – contradiction. ∎

The following is an adaption of Theorem III.3.10 in [180], with

$$p \vDash (Q,T) := \forall l.\ \mathsf{Forall}_2\,(\lambda x b.\ px \leftrightarrow b = \mathsf{true})\,Q\,l \to l \vDash T$$

for predicates $p\colon X \to \mathbb{P}$, query lists $Q\colon \mathbb{L}X$ and truth-tables T : tt.

Lemma 8.38. If $\mathcal{K} \preceq_{\mathsf{tt}} p$ there exists a function exceeding $\overline{p}$.

Proof. Let g be a tt-reduction from $\mathcal{K}$ to p. By Lemma 8.2 there is $\gamma\colon \mathbb{L}\mathbb{N} \to \mathbb{N}$ s.t. $\forall l x.\ \mathcal{W}_{\gamma l} x \leftrightarrow \neg(\lambda x.x \notin l) \vDash gx$.

Let $\mathsf{gen}_n : \mathbb{L}(\mathbb{L}\mathbb{N})$ contain exactly all duplicate-free lists l s.t. $\max l \le n$. We define $a_{n,i} := c(\mathsf{gen}_n[i])$ for $i < l^n$ and $a_{n,i} := c[]$ otherwise. Then $\lambda n.\ 1 + \max\big[\pi_1(ga_{n,i}) \mid i < 2^n\big]$ exceeds $\overline{p}$. To show this, let n be given and assume

$$(*) : \forall j.\ n < j < \max\big[\pi_1(ga_{n,i}) \mid i < 2^n\big] \to pj$$

We have to prove falsity. Note that $\neg\neg\exists.i < 2^n.\ \forall z.\ z \notin \mathsf{gen}_n[i] \leftrightarrow p^* z$ where $p^* z := (\neg\neg pz \wedge z \le n) \vee z > n$. Since we have to prove falsity, we can assume such an i. Now by $(*)$ we have $\forall j.n < j \in \pi_1(ga_{n,i}) \to pj$ since $\pi_1(ga_{n,i}) < \max\big[\pi_1(ga_{n,i}) \mid i < 2^n\big]$ follows from $i < 2^n$. Since for $x \le n$, $px \leftrightarrow p^* x$ by definition, we have $\forall x \in \pi_1(ga_{n,i}).\ px \leftrightarrow p^* x$. But now we obtain the following contradiction:

$$\begin{aligned} p \vDash ga_{n,i} &\leftrightarrow \mathcal{W}_{a_{n,i}}\, a_{n,i} \leftrightarrow \mathcal{W}_{\gamma(\mathsf{gen}_n[i])}\, a_{n,i} \\ &\leftrightarrow \neg(\overline{\mathsf{gen}_n[i]} \vDash ga_{n,i}) \leftrightarrow \neg(p^* \vDash ga_{n,i}) \\ &\leftrightarrow \neg(p \vDash ga_{n,i}) \end{aligned}$$ ∎

A predicate $p\colon \mathbb{N} \to \mathbb{P}$ is **hypersimple** if it is enumerable and its complement is non-finite and not majorised:

Def. hypersimple predicate

$$\mathsf{hypersimple}\,p := \mathcal{E}p \wedge \neg\mathcal{F}\overline{p} \wedge \neg\exists f.f \textit{ majorises } \overline{p}$$

Fact 8.39. Hypersimple predicates are not tt-complete.

Lemma 8.40. Given MP we have that hypersimple predicates are simple.

Proof. Direct by Corollary 8.37. ∎

8.7 Construction of a hypersimple predicate

We now construct and verify a hypersimple predicate, a result due to Post [189, §9]. We however follow Rogers [202, §8.1 Th. II], who presents a (more general) construction due to Dekker [52], defining a hypersimple predicate H_I for an arbitrary undecidable $I : \mathbb{N} \to \mathbb{P}$ with a strong, injective enumerator $E_I : \mathbb{N} \to \mathbb{N}$. By instantiating with $\mathcal{W}$ we obtain that $H_{\mathcal{W}}$ is hypersimple, and thus enumerable, undecidable, and tt-incomplete. The predicate $H_I : \mathbb{N} \to \mathbb{P}$ is defined as the so called "deficiency predicate" of I:

[52] Dekker. 1954. A theorem on hypersimple sets.

$$H_I x := \exists x_0 > x.\, E_I x_0 < E_I x$$

Lemma 8.41. $\overline{H_I}$ is non-finite.

Proof. By Corollary 3.34 it suffices to prove

$$\forall x. \neg\neg \exists y \geq x. \overline{H_I} y.$$

We use complete induction on $E_I x$. Given $x : \mathbb{N}$ assume $(*) : \neg\exists y \geq x. \overline{H_I} y$. The claim is negative, thus we can decide $H_I x$:

1. If $H_I x$, there exists $x_0 > x$ with $E_I x_0 < E_I x$. Hence, induction for x_0 yields $\neg\neg\exists y \geq x_0. \overline{H_I} y$ contradicting $(*)$.
2. If $\overline{H_I} x$ holds, x is an element as required. ∎

Lemma 8.42. If f majorises $\overline{H_I}$, then I is decidable.

Proof. Let f majorise $\overline{H_I}$. We prove that $g := \lambda x. x \in_{\mathbb{B}} \mathsf{map}\ E_I\ [0, \dots, f(\mathsf{S}\ x)]$ decides I, i.e. $\forall x.\ Ix \leftrightarrow gx = \mathsf{true}$. The direction from right to left is easy, since E_I enumerates I. For the converse direction let $E_I n = x$ for some n. We show $n \in [0, \dots, f(\mathsf{S}\ x)]$, which is stable. Thus let $n \notin [0, \dots, f(\mathsf{S}\ x)]$, i.e. $n > f(\mathsf{S}\ x)$. Since f majorises $\overline{H_I}$ and the goal is falsity, we can assume l with $\#l$, $|l| = \mathsf{S}\ x$ and $\forall y \in l. \overline{H_I} y \wedge y \leq f(\mathsf{S}\ x)$. Let $m := \max(\mathsf{map} E_I\, l)$. We have

1. $m \geq x$, since $|\mathsf{map} E_I l| = |l| > x$ and $\#(\mathsf{map} E_I l)$.
2. $m = E_I m_0$ for some $m_0 \in l$ and therefore $m_0 \leq f(\mathsf{S}\ x)$ and $\overline{H_I} m_0$ by the properties of l.

We now prove $H_I m_0$ to obtain a contradiction. We have $n > f(\mathsf{S}\ x) \geq m_0$ and $E_I n = x \leq m = E_I m_0$. By the injectivity of E_I, $E_I n = E_I m_0$ implies $n = m_0$ (a contradiction), such that we have $E_I n < E_I m_0$ as required. ∎

Theorem 8.43. There exists a hypersimple predicate.

Proof. $\mathcal{E}H_I$ follows by Corollary 4.20. $\lambda\langle c, x\rangle. \mathcal{W}_c x$ is an undecidable, strongly enumerable predicate I as assumed. ∎

Finally, we need to show H_I to be undecidable.

- First, we could show that H_I is simple. The general result that hypersimple predicates are simple (Lemma 8.40) seems to require MP, and the same holds for any obvious proof that H_I or $H_{\mathcal{W}}$ is simple. We thus do not consider this route.
- Secondly, we can give a direct proof of undecidability of H_I by showing that if H_I is decidable, then I is decidable.
- Thirdly, we can follow Rogers and give a Turing reduction $H_I \preceq_{\mathrm{T}} I$ [202, §8.1 Th. II]. We do so for our proposed notion of Turing reducibility in Section 9.5, again using MP.

Corollary 8.44. H_I is undecidable.

Proof. For a direct proof without assumptions, see [72] Appendix C. We will obtain the result as well in Corollary 9.20 using Turing reductions, but under the assumption of MP. ∎

[72] Forster, Jahn, and Smolka. 2021a. A Constructive and Synthetic Theory of Reducibility: Myhill's Isomorphism Theorem and Post's Problem for Many-one and Truth-table Reducibility in Coq.

Theorem 8.45. There exists an enumerable, undecidable, and tt-incomplete predicate.

Proof. By Fact 8.39, Corollary 8.44 and Theorem 8.43. ∎

8.8 Related Work

Bauer [10] also proves the existence of a synthetic simple set. Translated to our setting, in Bauer's definition simple predicates are predicates $p : X \to \mathbb{P}$ such that $\neg\mathcal{F}\overline{p} \land \forall q : X \to \mathbb{P}.\ \mathcal{G}q \to \exists x. qx \land px$. It seems like our definition and Bauer's are equivalent using MP, and that both directions of the equivalence rely on MP. We choose our definition to obtain a fully constructive proof of $\neg\mathcal{E}\overline{p}$ for any simple p. In the construction of a simple set, Bauer uses a partial selection function, where we use the total function $\psi : \forall c.\ (\exists x.\ \mathcal{W}_c x \land x > 2c) \to \mathbb{N}$, i.e. use an explicit proof argument as propositional guard. Both constructions rely on the enumerability of $\lambda c.\ \exists x.\ \mathcal{W}_c x \land x > 2c$, we use Lemma 4.26 to construct ψ, whereas Bauer uses Fact 4.62.

Our construction of both the simple and the hypersimple predicate is crucially based on the m-complete problem $\mathcal{K}$, where the non-enumerability of $\overline{\mathcal{K}}$ is shown directly using diagonalisation. Formally, we have $\overline{\mathcal{K}}c$ if φ_c does not have c in its range.

Other authors use different (m-complete) problems which are shown undecidable directly, and then center their analysis around these problems.

For instance, Turing [229] uses the problem "Given a Turing machine M and a symbol a, does M ever print a when started on the empty tape?", and Church [31] uses the problem "Given a λ-term t, does t have a normal form?".

[229] Turing. 1936. On computable numbers, with an application to the Entscheidungsproblem.

[31] Church. 1936. An unsolvable problem of elementary number theory.

Amongst authors concerned with simple sets, we have the following: Post [189] uses the problem "Given a Post system in normal form with basis B and a number n, does B produce n?". Davis [47] uses the problem "Given a Turing machine M and a string s, does M halt when started on a tape containing s?". Rogers [202] uses the problem "Given a number n, does the n-th μ-recursive function terminate on input n?".

[47] Davis. 1958. Computability and Unsolvability.

CHAPTER 9

Turing reducibility

One-one, many-one, and truth-table reducibility all have in common that their textbook definitions only rely on total computable functions in the chosen model of computation. For Turing reducibility, introduced by Turing [230], and dubbed like this by Post [189], the situation is different: Turing reducibility is based on oracle machines, an extension of the model of computation by oracles for arbitrary sets which can be queried arbitrarily often. Alternatively, Turing reducibility can be defined in terms of μ-recursive functionals [134, 47]. Intuitively, μ-recursive functionals as used for Turing reductions can *computably* transport potentially non-computable functions $\mathbb{N}\to\mathbb{B}$ (which can be thought of as oracles) to potentially non-computable functions $\mathbb{N}\to\mathbb{B}$.

[230] Turing. 1939. Systems of logic based on ordinals.

[189] Post. 1944. Recursively enumerable sets of positive integers and their decision problems.

[134] Kleene. 1952. Introduction to metamathematics.

[47] Davis. 1958. Computability and Unsolvability.

We use this intuition to define synthetic Turing reducibility, based on *continuous Turing functionals*. We follow an idea by Bauer [14] and define Turing functionals based on a two-layered approach: They consist of a (continuous) functional $r:(Y\rightsquigarrow\mathbb{B})\to(X\rightsquigarrow\mathbb{B})$ mapping functional relations $Y\rightsquigarrow\mathbb{B}$ to functional relations $X\rightsquigarrow\mathbb{B}$ factoring through a (then also continuous) computational core $r':(Y\rightharpoonup\mathbb{B})\to(X\rightharpoonup\mathbb{B})$.

[14] Bauer. 2020. Synthetic mathematics with an excursion into computability theory (slide set).

We also introduce two well-known refinements of Turing reducibility: Bounded Turing reducibility, which requires reduction functions to have a modulus of continuity, and total bounded Turing reducibility, which is equivalent to truth-table reducibility due to a result by Nerode [202, §9.6 Thm. XIX].

Lastly, we compare Bauer's definition of Turing reducibility to ours.

Three central theorems concerning Turing reducibility one finds in text-books are: Turing reductions differ from truth-table reducibility, proved by showing that the tt-incomplete hypersimple predicate H is Turing-reduction complete. We give a synthetic proof of this theorem in this chapter. The Kleene-Post theorem, stating that there are incomparable predicates under Turing-reducibility. The Friedberg-Muchnik theorem solving Post's problem for Turing reducibility, i.e. stating that there is an enumerable but undecidable predicate where the undecidability proof cannot be by Turing-reduction from the halting problem.

Outline We explain how the analytic definition of Turing reducibility based on μ-recursive functionals induces a synthetic definition of Turing reducibility in Section 9.1. We compare our definition with more naive definitions of Turing reducibility in Section 9.2.

In Section 9.3 and Section 9.4 we discuss bounded and total bounded Turing reducibility. In Section 9.5 we show that $\mathcal{W}$ reduces to the hypersimple predicate H. In Section 9.6 we compare Bauer's definition of Turing reducibility with ours.

Publications All results in this chapter are unpublished. The definition of Turing reducibility was conceived in joint work with Dominik Kirst.

9.1 Turing reducibility

Turing reducibility was introduced in Turing's PhD thesis [230], based on Turing machines

[230] Turing. 1939. Systems of logic based on ordinals.

which are extended with oracle calls to query non-computational functions in their execution. Kleene equivalently defines Turing reducibility via μ-recursive functionals [134]. A μ-recursive functional (in one function and one variable) F maps a partial function α and a number x partially to a number y. We suggestively write $F(\alpha)x \triangleright_\mu y$ to indicate the situation where the μ-recursive functional F is defined on α and x with value y. Notably, the input function α has not to be a partial *recursive* function, but if it is, $F(\alpha)$ is a μ-recursive function as well. This means that although F accepts non-computational input, it can be completely described by computable means, and in particular if α is computable, $F(\alpha)$ is.

[134] Kleene. 1952. Introduction to metamathematics.

The theorem that any partial μ-recursive functional F is both compact and monotonic is due to Kleene [134] and Davis [47]. We immediately use a constructivisation of the Kleene/Davis theorem, where

[47] Davis. 1958. Computability and Unsolvability.

1. F is compact, if whenever $F(\alpha)x \triangleright_\mu y$ there does *not not* exist a finite μ-recursive function $u \subset \alpha$ such that $F(u)x \triangleright_\mu y$.
2. F is monotonic, if whenever $F(\alpha)x \triangleright_\mu y$ for every μ-recursive function β such that $\alpha \subseteq \beta$ we have $F(\beta)x \triangleright_\mu y$.

We use this theorem by Kleene and Davis to define the synthetic notion of Turing functionals.

Def. Turing functional

A **Turing functional** is a functional $F\colon (Y \rightsquigarrow \mathbb{B}) \to (X \rightsquigarrow \mathbb{B})$ which is compact, monotonic, and factors through a computational core $F'\colon (Y \rightharpoonup \mathbb{B}) \to (X \rightharpoonup \mathbb{B})$.

More precisely, a Turing functional $F\colon (Y \rightsquigarrow \mathbb{B}) \to (X \rightsquigarrow \mathbb{B})$ ...

Def. compact functional

1. ... is **compact** if:

$$(\forall R\colon Y \rightsquigarrow \mathbb{B}.\forall x\colon X.\forall b\colon \mathbb{B}.\ FRx\,b \to \neg\neg \exists L : \mathbb{L}Y.\exists R_L : (Y \rightsquigarrow \mathbb{B}).$$
$$(\forall y\,b.\ R_L\,y\,b \to y \in L \wedge Ry\,b) \wedge FR_L\,x\,b)$$

Def. monotonic functional

2. ... is **monotonic** if:

$$\forall RR'.\ (\forall y\,b.\ Ry\,b \to R'\,y\,b) \to \forall x\,b.\ FRx\,b \to FR'\,x\,b$$

Def. computational core

3. ... factors through a **computational core** $F'\colon (Y \rightharpoonup \mathbb{B}) \to (X \rightharpoonup \mathbb{B})$ if:

$$\forall f\colon Y \rightharpoonup \mathbb{B}.\forall R\colon Y \rightsquigarrow \mathbb{B}.\ f \textit{ computes } R \to F'f \textit{ computes } FR$$

Def. computes

where a partial function $f\colon Z_1 \rightharpoonup Z_2$ **computes** a functional relation $R\colon Z_1 \rightsquigarrow Z_2$ if $\forall xy.\ Rxy \leftrightarrow f\,x \triangleright y$.

To simplify the formal treatment, we assume propositional extensionality Pext and functional extensionality Fext for the rest of the chapter. In particular, this implies extensionality of predicates.

Pext → Sec. 7.3, Page 70

Fext → Sec. 7.3, Page 70

$$\mathsf{PredExt} := \forall X\colon \mathbb{T}.\forall pq\colon X \to \mathbb{P}.\ (\forall x.\ px \leftrightarrow qx) \to p = q$$

Fact 9.1. Given propostional and functional extensionality we have PredExt.

Def. Turing reduction

A **Turing reduction** is a Turing functional which maps the characteristic relation of q to the characteristic relation of p. The characteristic relation of a predicate $p\colon X \to \mathbb{P}$ is the relation $\mathsf{char}\ p := \lambda xb.\ px \leftrightarrow b = \mathsf{true}$.

To sum everything up, a predicate $p : X \to \mathbb{P}$ is **Turing-reducible** to a predicate $q : Y \to \mathbb{P}$ if there exists a compact, monotonic Turing reduction: **Def.** Turing-reducible

$$
\begin{aligned}
p \preceq_{\mathrm{T}} q := \exists F : (Y \rightsquigarrow \mathbb{B}) \to (X \rightsquigarrow \mathbb{B}).\ & F(\mathsf{char}\ q) = \mathsf{char}\ p \wedge \\
& (\forall R : Y \rightsquigarrow \mathbb{B}. \forall x : X. \forall b : \mathbb{B}.\ FRxb \to \neg\neg \exists L : \mathbb{L}Y. \exists R_L : (Y \rightsquigarrow \mathbb{B}). \\
& \quad (\forall y b.\ R_L y b \to y \in L \wedge Ryb) \wedge FR_L xb) \wedge \\
& (\forall RR'.\ (\forall y b.\ Ryb \to R'yb) \to \forall x b.\ FRxb \to FR'xb) \wedge \\
& (\exists F' : (Y \rightharpoonup \mathbb{B}) \to (X \rightharpoonup \mathbb{B}). \forall f : Y \rightharpoonup \mathbb{B}. \forall R : Y \rightsquigarrow \mathbb{B}. \\
& \quad f \textit{ computes } R \to F' f \textit{ computes } FR)
\end{aligned}
$$

Without assuming extensionality axioms, the first condition would have to be spelled out in more detail. Furthermore, we would have to require that both F and F' map extensionally equal inputs to extensionally equal outputs, otherwise not even the following is provable:

Fact 9.2. Turing reducibility is a pre-order.

For other reducibility notions we considered, the degree of decidable predicates formed a minimum, i.e. decidable predicates reduced to every (non-trivial) predicate. Recall that decidable predicates are bi-enumerable, i.e. enumerable and their complement is, and that the converse ("Post's theorem") is equivalent to MP. For bounded Turing reducibility, the degree of bi-enumerable predicates is already a minimum:

Lemma 9.3. Let X be discrete and $p : X \to \mathbb{P}$. If $\mathcal{E}p$ and $\mathcal{E}\overline{p}$, then $p \preceq_{\mathrm{T}} q$.

Proof. Let f enumerate p and g enumerate $\overline{p}$. Then there exists a function $h : \mathbb{N} \rightharpoonup \mathbb{B}$ computing $\mathsf{char}\ p$. Define $FR := \mathsf{char}\ p$ and $F'f := h$. F is clearly compact and monotonic and factors through F'. ∎

Corollary 9.4. Let X be enumerable and discrete and $p : X \to \mathbb{P}$. Then $\mathcal{D}p \to p \preceq_{\mathrm{T}} q$.

Lemma 9.5. Given MP we have $p \preceq_{\mathrm{T}} q \to \mathcal{D}q \to \mathcal{D}p$.

Proof. Let $p \preceq_{\mathrm{T}} q$ via F and F'. Let f decide q. Then $F'(\lambda x.\ \mathsf{ret}\,(f\,x))$ computes $\mathsf{char}\ p\ \mathsf{true}$. Furthermore, $\forall x.\ \neg\neg(F'(\lambda x.\ \mathsf{ret}\,(f\,x))\downarrow)$ follows by case analysis on px, allowed since the proof goal is negative. By MP, we have $\forall x.\ F'(\lambda x.\ \mathsf{ret}\,(f\,x))\downarrow$. ∎

Since bi-enumerable predicates reduce to any predicate (in particular to arbitrary decidable predicates), a proof of backwards transport of decidability implies Post's theorem, meaning MP is necessary and sufficient for the previous theorem.

Lemma 9.6. MP if and only if $\forall pq : \mathbb{N} \to \mathbb{P}.\ p \preceq_{\mathrm{T}} q \to \mathcal{D}q \to \mathcal{D}p$.

Proof. The forward direction is Lemma 9.5. For the backward direction, we use Lemma 7.23 and prove that any bi-enumerable predicate $p : \mathbb{N} \to \mathbb{P}$ is decidable, i.e. $\mathcal{E}p \to \mathcal{E}\overline{p} \to \mathcal{D}p$, which suffices by Lemma 7.23 (4). It then suffices to prove that $p \preceq_{\mathrm{T}} (\lambda x.\top)$, which holds by bi-enumerability of p and Lemma 9.3. ∎

9.2 Naive Turing reducibility

The most naive synthetic definition of Turing reducibility would be to just focus on the core: A predicate $p : X \to \mathbb{P}$ is naively Turing reducible to $q : Y \to \mathbb{P}$ if there is a function $F : (Y \to \mathbb{B}) \rightharpoonup (X \to \mathbb{B})$ such that whenever f decides q, then $Ff \triangleright f'$ and f' decides p.

However, the fact that naive Turing reducibility is only defined for computable input means every predicate reduces to an undecidable predicate:

Lemma 9.7. If q is undecidable, then p naively Turing reduces to q.

Proof. Take the function $Ff := \mathsf{undef}$. Any f deciding q yields a contradiction. ∎

Under the presence of EA, this means that Post's problem for naive Turing reducibility is unsolvable, since $\mathcal{W}$ would reduce to any undecidable predicate, i.e. there is no enumerable, undecidable predicate incomplete under naive Turing-reducibility.

Thus, we model the reduction via functional relation transformers $F : (Y \rightsquigarrow \mathbb{B}) \to (X \rightsquigarrow \mathbb{B})$. Here, a naive approach would be to leave out compactness and monotonicity and just require that F factors though some F'. However, one can check in F whether the input is computable.

Lemma 9.8. Let $q : Y \to \mathbb{P}$ be undecidable, let $p : X \to \mathbb{P}$ and assume MP. Then there are functions $F : (Y \rightsquigarrow \mathbb{B}) \to (X \rightsquigarrow \mathbb{B})$ and $F' : (Y \rightharpoonup \mathbb{B}) \to (X \rightharpoonup \mathbb{B})$ such that $F(\mathsf{char}\ q) = \mathsf{char}\ p$ and $\forall (f : Y \rightharpoonup \mathbb{B})(R : (Y \rightsquigarrow \mathbb{B}))$. *computes* $R \to F'f$ *computes* FR.

Proof. Define $F'f := f$ and

$$FRxb := ((\exists f.\ f \textit{ computes } R) \wedge Rxb \leftrightarrow b = \mathsf{true}) \vee (\neg(\exists f.\ f \textit{ computes } R) \wedge px \leftrightarrow b = \mathsf{true})$$

Clearly, F' is the computational core of F, because if R is computable, $FR = R$.

If R is not computable, $FR = \mathsf{char}\ p$. Since q is undecidable, $\mathsf{char}\ q$ is not computable, and thus $F(\mathsf{char}\ q) = \mathsf{char}\ p$. ∎

9.3 Bounded Turing reducibility

A bounded Turing reduction is a Turing reduction with an additional function $B : X \to \mathbb{L}Y$ such that the reduction on input x only queries the oracle on elements of Bx. Given $F : (Y \rightsquigarrow \mathbb{B}) \to (X \rightsquigarrow \mathbb{B})$, this renders B a modulus of continuity of FR for all R, i.e. a modulus of continuity uniform in the relation argument.

Bounded Turing reducibility was initially introduced as *weak truth-table reducibility* by Friedberg and Rogers [86], see also the book by Odifreddi [180, p. 340].

[86] Friedberg and Rogers Jr. 1959. Reducibility and completeness for sets of integers.

[180] Odifreddi. 1992. Classical recursion theory: The theory of functions and sets of natural numbers.

A predicate $p : X \to \mathbb{P}$ is **bounded Turing-reducible** to $q : Y \to \mathbb{P}$ if there exists a bounded Turing reduction reducing the characteristic relation of q to the characteristic relation of p:

Def. bounded Turing-reducible

$$\begin{aligned} p \preceq_{\mathrm{bT}} q := \exists F : (Y \rightsquigarrow \mathbb{B}) \to (X \rightsquigarrow \mathbb{B}).\ & F(\mathsf{char}\ q) = \mathsf{char}\ p \wedge \\ & (\exists F' : (Y \rightharpoonup \mathbb{B}) \to (X \rightharpoonup \mathbb{B}).\forall f : Y \rightharpoonup \mathbb{B}.\forall R : (Y \rightsquigarrow \mathbb{B}). \\ & \quad f \textit{ computes } R \to F'f \textit{ computes } FR) \wedge \\ & (\exists B : X \to \mathbb{L}Y.\forall x : X.\forall RR' : (Y \rightsquigarrow \mathbb{B}). \\ & \quad (\forall y \in Bx.\forall b.\ Ryb \to R'yb) \to \forall b.\ FRxb \to FR'xb) \end{aligned}$$

Verbalised, the condition on B states that whenever two relations R and R' agree on Bx, then F treats them the same, meaning intuitively F only queries the oracle for the relation on inputs in Bx (or queries the oracle but ignores the result).

Note that we can leave out compactness and monotonicity, since they are implied by the bound function, which is used in the following proof:

Lemma 9.9. $p \preceq_{\mathrm{bT}} q \to p \preceq_{\mathrm{T}} q$

Proof. Any bounded Turing reduction F is a Turing reduction. The bound function B implies compactness (even without a double negation) and monotonicity of F. ∎

The converse direction is not provable because one cannot turn a proof of compactness and monotonity to a bound function. One might be able to give a synthetic proof that the notions are distinct base on EA.

Fact 9.10. Bounded Turing reducibility is a pre-order.

Similar to Turing reducibility, MP is necessary and sufficient for the transport of decidability.

Fact 9.11. MP if and only if $p \preceq_{\mathrm{bT}} q \to \mathcal{D}q \to \mathcal{D}p$.

9.4 Total bounded Turing reducibility

A Turing reduction from p to q only has to produce a total relation as output when it is given the characteristic relation for q as input. This condition can be strengthened such that any total input relations is mapped to a total output relation, resulting in the concept of total bounded Turing reducibility.

Formally, a predicate $p\colon X \to \mathbb{P}$ is **total bounded Turing-reducible** to a predicate $q\colon Y \to \mathbb{P}$ if there exists a bounded Turing reduction which reduces the characteristic relation of q to the characteristic relation of p and which transports total inputs to total outputs:

Def. total bounded Turing-reducible

$$
\begin{aligned}
p \preceq_{\mathrm{tbT}} q := \exists F\colon (Y \rightsquigarrow \mathbb{B}) \to (X \rightsquigarrow \mathbb{B}).\ & F(\mathsf{char}\ q) = \mathsf{char}\ p \wedge \\
& (\forall R\colon (Y \rightsquigarrow \mathbb{B}).\ (\forall y.\exists b.Ryb) \to (\forall x.\exists b.FRxb)) \wedge \\
& (\exists F'\colon (Y \rightharpoonup \mathbb{B}) \to (X \rightharpoonup \mathbb{B}).\forall (f : Y \rightharpoonup \mathbb{B})(R : (Y \rightsquigarrow \mathbb{B})).\\
& \quad f \textit{ computes } R \to F'f \textit{ computes } FR) \wedge \\
& (\exists B\colon X \to \mathbb{L}Y.\forall (x : X)(RR' : (Y \rightsquigarrow \mathbb{B})).\\
& \quad (\forall y \in Bx.Ry = R'y) \to FRx = FR'x)
\end{aligned}
$$

Fact 9.12. $p \preceq_{\mathrm{tbT}} q \to p \preceq_{\mathrm{bT}} q$

Total bounded Turing reductions implicitly contain a total computational core.

Fact 9.13. Let $F\colon (Y \rightsquigarrow \mathbb{B}) \to (X \rightsquigarrow \mathbb{B})$, $F'\colon (Y \rightharpoonup \mathbb{B}) \to (X \rightharpoonup \mathbb{B})$, and $B\colon X \to \mathbb{L}Y$ be given such that

- $\forall R\colon (Y \rightsquigarrow \mathbb{B}).(\forall y.\exists b.Ryb) \to (\forall x.\exists b.rRxb)$,
- $\forall (f\colon Y \rightharpoonup \mathbb{B})(R\colon (Y \rightsquigarrow \mathbb{B})).f$ *computes* $R \to F'f$ *computes* FR,
- $\forall (x\colon X)(RR'\colon (Y \rightsquigarrow \mathbb{B}))$. $(\forall y \in Bx.Ry = R'y) \to rRx = rR'x$.

Then there exists a function $F'' : : (Y \to \mathbb{B}) \to (X \to \mathbb{B})$ such that

1. $\forall f : Y \to \mathbb{B}.\, F(\lambda x\, b.f\, x = b)x\, b \leftrightarrow F''f\, x = b$
2. $\forall x(f\, f' : Y \to \mathbb{B}).\, (\forall y \in Bx.f\, y = f'y)F''f\, x = F''f'x$

This total part can be used to transport decidability backwards fully constructively.

Lemma 9.14. If $p \preceq_{\mathrm{tbT}} q$ and q is decidable, then p is decidable.

Proof. Let f decide q. Then $F''f$ decides p. ∎

As expected, truth-table reducibility implies total bounded Turing reducibility. We need MP for the proof.

Lemma 9.15. $\mathsf{MP} \to p \preceq_{\mathrm{tt}} q \to p \preceq_{\mathrm{tbT}} q$

Proof. Let $f' : X \to \mathbb{L}Y \times \mathsf{truthtable}$ be a truth-table reduction from p to q.

One can define a function $f : X \to \mathbb{L}\mathbb{P} \to \mathbb{P}$ s.t.using MP

1. $\forall l : \mathbb{L}\mathbb{B}.\forall x : X.\, l \vDash \pi_2(f'x) \leftrightarrow f\, x(\mathsf{map}(\lambda b.\, b = \mathsf{true})l)$
2. $\forall x.\, px \leftrightarrow f\ x\ (\mathsf{map}q(\pi_1(f'x)))$

The function f can be seen as the extension of $\lambda x l.l \vDash \pi_2(f'x)$ to propositions, or vice versa $\lambda x l.l \vDash \pi_2(f'x)$ is the computational core of f.

Now we define

$$FRx\, b := \exists L : \mathbb{L}\mathbb{P}.\, \mathsf{Forall}_2(\lambda x P.\, (P \to Rx\mathsf{true})\ \wedge (\neg P \to Rx\mathsf{false}))\, (\pi_1(f\, x))\, L$$

$$\wedge f\, x L \leftrightarrow b = \mathsf{true}$$

$$F'f\, x := \mathsf{map}_{\mathsf{part}}\ g(\pi_1(f'x)) \gg\!= \lambda l.\, \mathsf{ret}\,(\pi_2(f'x)L)$$

$$Bx := \pi_1(f'x)$$

where $\mathsf{map}_{\mathsf{part}} : (X \rightharpoonup Y) \to \mathbb{L}X \rightharpoonup \mathbb{L}Y$ s.t.if f computes R, $\mathsf{map}_{\mathsf{part}}\, f$ computes $\mathsf{Forall}_2 R$. ∎

Rogers attributes a proof that total bounded Turing reductions have a bound and thus are truth-table reductions to Anil Nerode [202, §9.6, Thm. XIX]. We prove the latter part as well and show that total bounded Turing reductions are in fact truth-table reductions, crucially relying on the total computational core from above.

Theorem 9.16. Let Y be discrete. If $p \preceq_{\mathrm{tbT}} q$, then $p \preceq_{\mathrm{tt}} q$.

Proof. Let F be the bounded total Turing reduction, F' its computational core, $B : X \to \mathbb{L}Y$ their bound function, and $F'' : : (Y \to \mathbb{B}) \to (X \to \mathbb{B})$ its total computational core by Fact 9.13.

For input x, we define the queries of the truth-table reduction to be Bx. For the truth-table, given a list of answers l for Bx, we feed F'' with a function mapping every element of Bx to its answer defined in l, i.e. as $\lambda l.F''(f\, l)$ where

$$f\, l\, y := \textbf{if}\ \mathsf{pos}\, y\, (Bx)\ \textbf{is}\ \mathsf{Some}\, n\ \textbf{then if}\ l[n]\ \textbf{is}\ \mathsf{Some}\, b\ \textbf{then}\ b\ \textbf{else}\ \mathsf{false}\ \textbf{else}\ \mathsf{false}$$ ∎

9.5 The hypersimple predicate H_I Turing-reduces to I

Recall that we defined the hypersimple predicate $H_I : \mathbb{N} \to \mathbb{P}$ as the deficiency predicate of a strongly enumerable predicate $I : \mathbb{N} \to \mathbb{P}$. That is, if E_I is an injective, strong enumerator of I ($\forall x.\, Ix \leftrightarrow \exists n. E_I n = x$), we have

$$H_I x := \exists x_0 > x.\, E_I x_0 < E_I x$$

Algorithmically, we can decide Iz given a partial function $f : \mathbb{N} \rightharpoonup \mathbb{B}$ computing H_I as follows: We search for x such that $f\,x \triangleright \mathsf{false}$ and $E_I x > z$, i.e. $\neg H_I x$. Such an x does (not not) exists because $\overline{H_I}$ is non-finite. Then Iz holds if and only if $z \in [E_I 0, \dots E_I(x+1)]$.

Formally:

Fact 9.17. If $p : X \to \mathbb{P}$ is non-finite and $f : X \to \mathbb{N}$ is injective, then $\neg\neg\exists x.\, px \wedge f\,x \geq z \wedge \forall y.\, py \to f\,y \geq z \to f\,x \leq f\,y$.

Fact 9.18. If $\neg H_I x$ and $E_I x > z$, then $Iz \leftrightarrow [E_I 0, \dots, E_I(x+1)]$.

Theorem 9.19. Assume MP and let I be a strongly enumerable undecidable predicate. Then $I \preceq_{\mathrm{T}} H_I$.

Proof. We define

$$\begin{aligned} F' f z := \mu(\lambda x.(f\,x) \gg\!= \lambda b.\mathbf{if}\ \neg_{\mathbb{B}} b \wedge E_I x > z\ \mathbf{then}\ \mathsf{ret}\,\mathsf{true}\ \mathbf{else}\ \mathsf{ret}\,\mathsf{false}) \gg\!= \\ \lambda x.\mathsf{ret}\,(z \in_{\mathbb{B}} [E_I 0, \dots, E_I(x+1)]) \end{aligned}$$

$$\begin{aligned} FRzb := \neg\neg\exists x. Rx\mathsf{false} \wedge E_I x > z \wedge \\ (\forall x' < x. \neg\neg(Rx'\mathsf{true} \vee (Rx'\mathsf{false} \wedge E_I x' \leq z))) \wedge \\ b = \mathsf{true} \leftrightarrow z \in [E_I 0, \dots, E_I(x+1)] \end{aligned}$$

Note that we need the second line in F to ensure that the unbounded search in F' indeed computes F.

To prove that F' computes F we need MP, which allows us to prove $FRzb \leftrightarrow \neg\neg F' f z \triangleright b$ by Fact 7.24. Functionality, compactness, and monotonicity of F are technical to prove, but overall straightforward.

We again use MP to prove the correctness of F, which allows us to prove $F(\mathsf{char}\, H_I)zb \leftrightarrow \neg\neg Iz$ by Lemma 7.23 (1), since I is enumerable.

The direction from left to right is immediate from Fact 9.18. For the direction from right to left, let $b = \mathsf{true} \leftrightarrow Iz$. Let x be obtained for H_I and E_I from Fact 9.17. Then x fulfils the claim by Fact 9.18. ∎

Corollary 9.20. Assume MP and let I be a strongly enumerable undecidable predicate. Then I is undecidable.

9.6 Related work

While we define Turing reducibility with a focus on functional relations and computing functions, Bauer defines Turing reducibility based on predicates and enumerability [14]. We trans-

[14] Bauer. 2020. Synthetic mathematics with an excursion into computability theory (slide set).

late his definition to type theory by modelling $\mathcal{P}(X)$ as $X \to \mathbb{P}$.

The type of oracles oracle is defined as the type of pairs $(S_0 : X \to \mathbb{P}, S_1 : X \to \mathbb{P})$ such that $\forall x.\, S_0 x \to S_1 x \to \bot$. Oracles are in direct correspondence to total functional relations via the following two translations:

$$R_{(S_0,S_1)} := \lambda x\, b.\, (S_0 x \leftrightarrow b = \mathsf{true}) \wedge (S_1 x \leftrightarrow b = \mathsf{false}) \qquad O_R := (\lambda x.\, R\ x\ \mathsf{true}, \lambda x.\, R\ x\ \mathsf{false})$$

Fact 9.21. $R_{O_R} = R$ and $O_{R_O} = O$.

Fact 9.22. Let X be discrete. If f_0 and f_1 enumerate S_0 and S_1, respectively, then $\lambda x.\mu(\lambda n.\mathsf{ret}\,(f_0 n =_{\mathbb{B}} \mathsf{Some}\, x \vee_{\mathbb{B}} f_1 n =_{\mathbb{B}} \mathsf{Some}\, x)) \gg\!= \lambda n.\, f_0 n =_{\mathbb{B}} \mathsf{Some}\, x$ computes $R_{(S_0,S_1)}$.

Fact 9.23. Let g enumerate X. If f computes R, then S_0 of O_R is enumerated by the following function (and S_1 has a similar enumerator):

$$\lambda\langle m, n\rangle.\ \textbf{if}\ g m\ \textbf{is}\ \mathsf{Some}\, x\ \textbf{then if}\ \mathsf{seval}\,(f\,x)\ n\ \textbf{is}\ \mathsf{Some}\,\mathsf{true}\ \textbf{then}\ \mathsf{Some}\, x\ \textbf{else}\ \mathsf{None}\ \textbf{else}\ \mathsf{None}$$

We denote Bauer's definition of Turing reducibility with $\preceq_{\mathrm{B}}$.

$$\begin{aligned} p \preceq_{\mathrm{B}} q := \exists F : \mathsf{oracle} \to \mathsf{oracle}.\ & F(q, \overline{q}) = (p, \overline{p}) \wedge \\ & \exists e_0 : (\mathbb{N} \to \mathbb{O}X) \to (\mathbb{N} \to \mathbb{O}X) \to (\mathbb{N} \to \mathbb{O}X) \\ & \exists e_1 : (\mathbb{N} \to \mathbb{O}X) \to (\mathbb{N} \to \mathbb{O}X) \to (\mathbb{N} \to \mathbb{O}X) \\ & \quad \forall f_0 f_1 S_0 S_1.\ f_0 \textit{ enumerates } S_0 \to f_1 \textit{ enumerates } S_1 \to \\ & \quad \textbf{let}\ (S_0', S_1') := F(S_0, S_1)\ \textbf{in} \\ & \qquad e_0\ f_0\ f_1 \textit{ enumerates } S_0' \wedge e_1\ f_0\ f_1 \textit{ enumerates } S_1' \end{aligned}$$

As for our definition of Turing reducibility, Bauer's notion has to be extended with a condition that F is continuous, e.g. that it preserves directed suprema. We leave an analysis of which continuity conditions to choose for the two otherwise equivalent definitions of Turing reducibility to future work.

9.7 Future work

Three directions for future work are apparent:

First, it would be interesting to study the exact connection to Bauer's notion of Turing reducibility. It might be possible to find a formalisation of the preservation of concrete suprema in a way that our and Bauer's notion become exactly equivalent, but since interesting properties of Turing reducibility rely on MP, assuming MP for an equivalence proof also seems unproblematic.

[172] Muchnik. 1963. On strong and weak reducibility of algorithmic problems.

[85] Friedberg. 1957. Two Recursively enumerable sets of incomparable degrees of unsovlability (solution of Post's problem, 1944.

[144] Kučera. [n. d.]. An alternative, priority-free, solution to Post's problem.

[127] Kirst and Hermes. 2021a. Synthetic Undecidability and Incompleteness of First-Order Axiom Systems in Coq.

Secondly, it would be interesting to settle Post's problem synthetically: to formalise the priority method due to Friedberg and Muchnik [172, 85] and establish that there is a predicate which is enumerable, undecidable, but Turing-reducibility incomplete. Alternatively, Kučera [144] solves Post's problem without the priority method by using Peano arithmetic. A synthetic machine-checked version of this proof could work on the synthetic treatment of Peano arithmetic by Kirst and Hermes [127].

Thirdly, as an intermediate step for exploration, it would be interesting to formalise the Kleene-Post theorem [136] and establish that there are two (non-enumerable) predicates which are not comparable under Turing reducibility.

Part II

Models of computation

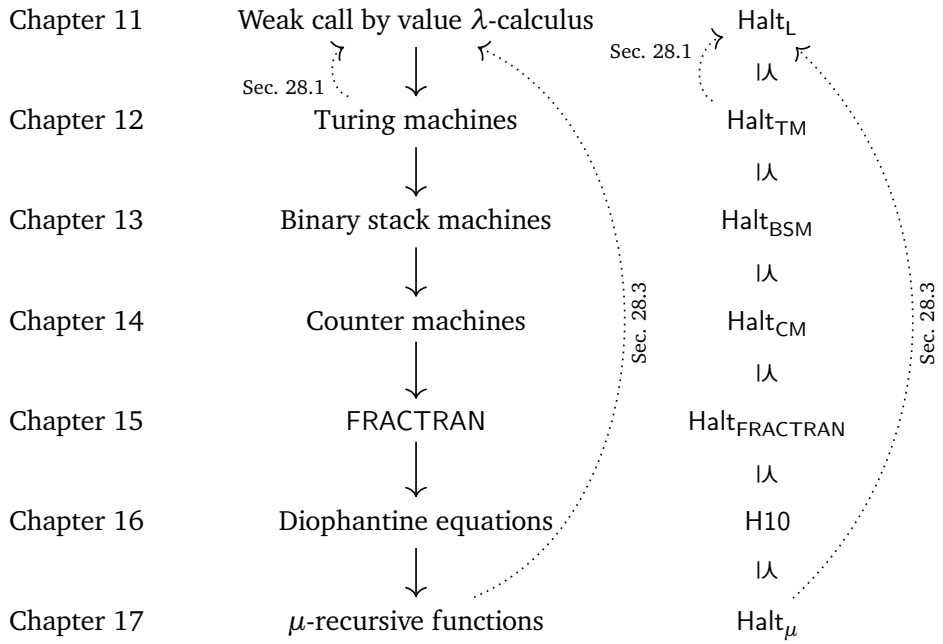


Figure 10.1.: Outline of Part II.

CHAPTER 10

Introduction: Models of computation

In this part of the thesis, we give machine-checked Turing-equivalence proofs for several models of computation in the Coq proof assistant. We show that the weak call-by-value λ-calculus L, multi-tape Turing machines, single-tape Turing machines, binary single-tape Turing machines, binary stack machines, counter machines, FRACTRAN, Diophantine equations, and μ-recursive functions are all equivalent.

We are not aware of a previous complete and uniform formalisation of all mentioned models. Besides this, the contribution of this part of the thesis is two-fold.

First, we identify levels of abstraction sufficient to implement and verify non-trivial constructions and algorithms in the respective models. We contribute a verification framework for Turing machines which allows giving and verifying algorithms in the style of a register-based while-language, and an extraction framework for the weak call-by-value λ-calculus L which automatically yields verified L-terms computing simply-typed functions.

Secondly, we invest the proof engineering effort to fully mechanise all simulations, in total comprising 52k lines of collaborative Coq code (for more details see Section 10.3) We observe that often folklore parts or routine chores take the most proof engineering time.

We first discuss the challenges involved in machine-checked proofs in comparison to proof sketches and formalised proofs, before giving a mathematical outline of the shape of simulation theorems we prove.

Informal proofs vs. formalised proofs vs. machine-checked proofs

The first work on a Turing-equivalence proof for a model of computation was started by Church, Kleene, and Rosser, who proved in the early 30s that the λ-calculus and μ-recursive functions are equivalent. The first published such proof is due to Kleene [131].

Shortly after, Turing published his result that the λ-calculus and his (Turing) machines are equivalent [229]. Turing sketched the *high-level ideas for the constructions* necessary for the two directions of the equivalence. He neither spelled out μ-recursive functions or λ-terms involved, nor gave an actual proof why the functions and terms are correct.

In contrast, Kleene's proofs are more detailed, but still only remain *informal proofs*: for instance, generalised invariants necessary for successful induction are not spelled out. Kleene's direct equivalence proof of Turing machines and μ-recursive functions [134], without mention of the λ-calculus, is given on a similar level.

More detailed equivalence proofs stem from the 21st century and investigations into the *invariance thesis*, asserting that reasonable models of computation can simulate each other with polynomial time and constant factor space overhead [208]. Dal Lago and Martini [149] give a detailed proof that Turing machines can simulate (a form of) the λ-calculus, arguing about the content of tapes at different points of the simulation. Dal Lago and Accatoli [148] give a formalised proof that any λ-calculus can simulate Turing machines on paper, spelling out all details.

[131] Kleene. 1936. λ-definability and recursiveness.

[229] Turing. 1936. On computable numbers, with an application to the Entscheidungsproblem.

[134] Kleene. 1952. Introduction to metamathematics.

[47] Davis. 1958. Computability and Unsolvability.

[208] Slot and van Emde Boas. 1984. On Tape Versus Core; An Application of Space Efficient Perfect Hash Functions to the Invariance of Space.

[149] Lago and Martini. 2008. The weak lambda calculus as a reasonable machine.

[148] Lago and Accattoli. 2017. Encoding Turing Machines into the Deterministic Lambda-Calculus.

In general, textbooks and research papers tend to give proofs for interesting novel parts, while omitting proofs for folklore parts. This technique is omnipresent in theoretical computer science and mathematics: The folklore parts of results are only sketched or are entirely left out, while the interesting parts are formalised and detailed proofs are given.

When mechanising a result in an interactive theorem prover, both aspects form their own challenges: Starting with a formal proof, only missing details of an argument have to be recovered. Starting with folklore results, first a proof has to be found, then formalised, then mechanised, and each individual step can prove challenging.

Thus, for machine-checked proofs it can happen that the folklore parts are harder to mechanise than the non-folklore parts, since the amount of missing details can vary greatly.

Simulation proofs

That two models of computation can simulate each other is not a formally defined notion. In fact, we will usually prove one theorem and deduce two corollaries to establish the simulation of a model M_1 in a model M_2. All three statements can be argued to express that M_2 can simulate M_1.

For all models we define an evaluation relation $\_(\_) \triangleright_M \_ : P_M \to I_M \to O_M \to \mathbb{P}$. The program type P_M, input type I_M, and output type O_M differ for different M. For some models $I_M = O_M$, e.g. for Turing or counter machines. For others like the λ-calculus, there is no native input, i.e. $I_M = \mathbb{1}$, and the output are programs, i.e. $O_M = P_M$.

The theorem we prove for a simulation of M_1 on M_2 is either a *compilation* or *interpretation* theorem. Compilation theorems establish translation functions $\delta_P : P_{M_1} \to P_{M_2}$, $\delta_I : I_{M_1} \to I_{M_2}$, and $\delta_O : O_{M_1} \to O_{M_2}$ such that if $c(x) \triangleright_{M_1} v$ then $\delta_P c\ (\delta_I x) \triangleright_{M_2} \delta_O v$. Interpretation theorems establish a single program I of M_2 and encodings $\gamma : P_{M_1} \to I_{M_1} \to I_{M_2}$ and $\gamma' : O_{M_1} \to O_{M_2}$ such that if $c(x) \triangleright_{M_1} v$ then $I(\gamma c x) \triangleright_{M_2} \gamma' v$. In both cases there are further requirements for the full theorem which we omit here.

Interpretation theorems directly imply compilation theorems by use of the S_n^m theorem (i.e. partial application) for M_2. Vice versa, compilation theorems yield interpretation theorems by use of a universal machine for M_2.

We choose between proving a compilation or interpretation theorem depending on the models M_1 and M_2 we consider. Note that both interpretation and compilation theorems have encodings of programs and data of M_1 as programs and data of M_2 built into the statement. Composing compilation theorems with each other or with interpretation theorems is possible in principle, but results in complicated composed encoding functions.

For instance, encoding natural numbers (the input of counter machines) on Turing machines is easy. Composing the interpretation theorem of μ-recursive functions with the interpretation theorem of L on Turing machines results in an interpretation theorem converting μ-recursive functions to Turing machines, which however work on the Turing machine-encoding of the L-encoding of natural numbers. Thus, using such composed theorems is unfeasible in practice, and we always deduce two corollaries which are uniform for all M_1 and M_2. In contrast to compilation or interpretation theorems, composing the corollaries does not alter input encodings and we do not need encodings of one model in the other for the specification of the corollaries.

The first corollary is a *simulation result w.r.t. halting*. It states that the halting problem of M_1 many-one reduces to the halting problem of M_2, i.e. that the problem of determining whether

a program in M_1 halts on some input data can be turned into the problem of determining whether a program in M_2 halts on some input data. Formally, we always prove

$$\mathsf{Halt}_{M_1} \preceq_m \mathsf{Halt}_{M_2} \quad \textit{where} \quad \mathsf{Halt}_M(c : P_M, x : I_M) := \exists v : O_M.\, c(x) \triangleright v.$$

We use the definition of many-one reducibility $\preceq_m$ for $p : X \to \mathbb{P}$ and $q : Y \to \mathbb{P}$ from Part I:

$\preceq_m$ → Sec. 5.1, Page 46

$$p \preceq_m q := \exists f : X \to Y.\, \forall x.\, px \leftrightarrow q(f\,x)$$

Note how the halting problem corollary is concerned with halting only, and not with the concrete return value of a computation. As such it can be used to transport semi-decidability results: If a problem p is semi-decidable in M_1, and $\mathsf{Halt}_{M_1} \preceq_m \mathsf{Halt}_{M_2}$, then p is also semi-decidable in M_2. This result comes for free since semi-decidability is only concerned with the halting of a program on some input. To prove a similar result for the decidability of problems, one could still rely on halting only by assuming MP and proving Post's theorem that a problem p is decidable if both p and its complement are semi-decidable for every M_2 individually. However to state for example that a many-one reduction proof via a function $f : \mathbb{N} \to \mathbb{N}$ can be transported between equivalent models, it is crucial to consider the output of a computation as well.

MP → Sec. 3.1, Page 20

As second corollary, we thus always deduce a *simulation results w.r.t. computability*, considering both input and output. The different models of computation we consider have different types of input and output. Since we aim at uniform statements of the simulation results w.r.t. computability, we formulate them for k-ary relations on natural numbers. More precisely, we define for all models what it means to compute a k-ary (functional) relation on natural numbers. In Chapter 9 we have defined that a relation $R : X \to Y \to \mathbb{P}$ is computed by a partial function $f : X \rightharpoonup Y$ if $\forall x\, y.\, Rxy \leftrightarrow f\,x \triangleright y$. To define when a function in a model of computation computes a relation, we use a similar approach.

For every model M we define encoding functions $\overline{\cdot} : \mathbb{N} \to O_M$ and $\overline{\cdot} : \mathbb{N}^k \to I_M$ and define that a program c computes a k-ary relation $R : \mathbb{N}^k \to \mathbb{N} \to \mathbb{P}$ if the following hold:

1. $\forall n_1 \dots n_k m : \mathbb{N}.\, R(n_1, \dots, n_k)m \leftrightarrow c(\overline{n_1, \dots, n_k}) \triangleright \overline{m}$
2. $\forall n_1 \dots n_k : \mathbb{N}. \forall v : D.\, c(\overline{n_1, \dots, n_k}) \triangleright v \to \exists m. v = \overline{m}$

Condition (1) is similar to the condition for partial functions. Condition (2) can be read as well-typedness condition on M. This becomes necessary because evaluation in models is untyped, and M could terminate with ill-formed output not corresponding to a natural number. In fact, for every considered model, condition (2) can be proved superfluous: Any program M' can be turned into a well-typed program M satisfying condition (2), but this requires writing a program which detects whether an output is well-formed, which is tedious. We thus include the condition explicitly to obtain more elegant proofs.

Equivalently, if M has a deterministic evaluation relation then c computes R if and only if

1. $\forall n_1 \dots n_k m : \mathbb{N}.\, R(n_1, \dots, n_k)m \to c(\overline{n_1, \dots, n_k}) \triangleright \overline{m}$
2. $\forall n_1 \dots n_k : \mathbb{N}. \forall v : M. c(\overline{n_1, \dots, n_k}) \triangleright v \to \exists m. R(n_1, \dots, n_k)m.$

Note that it is crucial to consider relations rather than functions $\mathbb{N}^k \to \mathbb{N}$ to transport *un*computability proofs between models. Since CIC only allows the definition of computable functions, an uncomputable relation cannot be defined as a function.

10.1 Outline

A graphical outline of this part can be found in Figure 10.1.

The translation from L to Turing machines is the most intricate: L is a higher-order language, where terms are tree-like and contain binders, while Turing machines are the most low-level model we consider. We thus narrow the gap from two directions: We introduce a stack machine semantics for L, which has a linearised encoding of terms and no binder structure. We then introduce a verification framework for Turing machines, allowing us to program Turing machines via a shallowly embedded imperative instruction language treating tapes as registers. We assemble all ingredients to a Turing machine constituting a verified L-interpreter. The compiler from Turing machines to binary stack machines is novel. It uses a compilation from multi-tape to single-tape Turing machines, and a novel compilation from arbitrary Turing machines to binary Turing machines. We introduce simple binary Turing machines as intermediate model, and give a direct compiler from simple binary Turing machines to binary stack machines. The compilation theorem for binary stack machines to counter machines was devised and implemented by Dominique Larchey-Wendling [81]. We recap the compilation and linking phases involved and use the theorem to deduce a simulation w.r.t. computability theorem in addition to the halting problem reduction. The translation from counter machines to FRACTRAN is a direct compiler, following the initial proof idea by Conway [35]. The translation from FRACTRAN to Diophantine equations amounts to proving that the reflexive, transitive closure of the FRACTRAN transition relation is Diophantine. This proof was carried out by Larchey-Wendling [81], we only summarise the main points. A preliminary interpretation theorem for Diophantine equations using μ-recursive functions is also due to Larchey-Wendling [153]. We extend it to prove a simulation theorem w.r.t. computability. Finally, we close the loop by an interpretation theorem for μ-recursive functions using L in Section 28.3.

[35] Conway. 1987. Fractran: A simple universal programming language for arithmetic.

10.2 Related work

History of models of computation

The notion of *intuitive calculability* of functions[1] is already prevalent in early 20th century math: The "process using finitely many operations"[2] asked for in Hilbert's tenth problem regarding the solvability of Diophantine equations is exactly such an intuitively calculable function. The *Entscheidungsproblem*, which asked for an intuitively calculable function to determine the validity of a formula in first-order logic, was identified by Hilbert and Ackermann to be the "central problem of mathematical logic"[3] in 1928 [110]. Hilbert's dream of a negation-complete logic with intuitively calculable provability relation would have entailed a universal decider for every problem expressible in logic.

[110] Hilbert and Ackermann. 1928. Grundzüge der theoretischen Logik.

As hinted by the name, intuitive calculability is not a formal notion. Already in the 19th century it was however common to formally define closed sub-classes of intuitively calculable functions. For instance, primitive recursive functions were already defined by Dedekind [50]. The term *primitively recursive* was coined by Péter after Ackermann proved that not every intuitively calculable function is primitively recursive [1].

[1] Ackermann. 1928. Zum Hilbertschen Aufbau der reellen Zahlen.

[1] often also called *effectively calculable* or *intuitively recursive* functions.

[2] "Verfahren [...] durch endlich viele Operationen" in the original German formulation.

[3] "das Entscheidungsproblem muß als das Hauptproblem der mathematischen Logik bezeichnet werden"

In 1933, Gödel, inspired by earlier work of Herbrand [107, 108], defined the μ-recursive functions (extending primitively recursive functions by unbounded minimisation) [93]. He explicitly posed the question whether μ-recursive functions comprise all intuitively calculable functions, but remains unconvinced that an answer can be positive.

In 1932, Church proposed the λ-calculus as system for logic [30], which was proved logically inconsistent by Kleene and Rosser [137]. However, Church, Kleene, and Rosser proved that λ-definability of a function is equivalent to μ-recursiveness, see for instance the paper published by Kleene [131]. The equivalence proof and the work by Kleene and Rosser on proving larger and larger classes of intuitively calculable functions to be λ-definable motivated Church to take up the question by Gödel, and announce μ-recursiveness as *definition* of intuitive calculability [31]. The thesis claiming that every intuitively calculable functions is μ-recursive (and equivalently λ-definable), later became known as *Church's thesis*. Based on a diagonal argument, Church proved that the characteristic function of the halting problem is not λ-definable and by his thesis thus not intuitively calculable. As a consequence, there is no λ-definable and by Church's thesis no intuitively computable decision function for the Entscheidungsproblem. Gödel however states that he is "not at all convinced that [Church's] concept of recursion comprised all possible recursions" in a letter to Davis [46].

Simultaneously to the development of Church's thesis, Turing devised machines which later became to be known as Turing machines. Turing machines are a form of finite state machines operating on an infinite tape [229]. Turing provided a proof that every function which is *mechanically calculable* by a human *computor* is computable by a Turing machine. Since mechanical calculability and intuitive calculability can be identified, this amount to *Turing's thesis* that every intuitively calculable function is Turing machine-computable. Also using diagonalisation, Turing proved that the problem of determining whether a Turing machine ever prints a given symbol[4] is not Turing-computable, and deduces that the Entscheidungsproblem is also not Turing-computable. By Turing's thesis, this means that a decider for the Entscheidungsproblem is not intuitively calculable.

After learning about Church's thesis, Turing sketches in an appendix that Turing machines and the λ-calculus are equivalent, thus making Church's thesis and Turing's thesis equivalent – the terminology "Church-Turing thesis" was coined by Kleene [134]. Only Turing's argument convinces Gödel: He described it as "most satisfactory" and "correct [...] beyond any doubt" [94].

[107] Herbrand. 1930. Les bases de la logique Hilbertienne.

[108] Herbrand. 1932. Sur la non-contradiction de l'Arithmétique.

[93] Gödel. 1934. On formally undecidable propositions of Principia Mathematica and related systems.

[30] Church. 1932. A set of postulates for the foundation of logic.

[137] Kleene and Rosser. 1935. The inconsistency of certain formal logics.

[131] Kleene. 1936. λ-definability and recursiveness.

[31] Church. 1936. An unsolvable problem of elementary number theory.

[46] Davis. 1982. Why Gödel Didn't Have Church's Thesis.

[229] Turing. 1936. On computable numbers, with an application to the Entscheidungsproblem.

[134] Kleene. 1952. Introduction to metamathematics.

Machine-checked formalisations

Machine-checked formalisations of models of computation are relatively frequent in the literature. Most of this related work does not translate between models of computation, with one notable exception: Xu, Zhang, and Urban [239] mechanise Turing machines, abacus machines (a form of counter machines), and μ-recursive functions in the higher order logic of the Isabelle theorem prover. They present machine-checked translations of μ-recursive functions to abacus machines, and of abacus machines to Turing machines, employing a Hoare logic for Turing machines. Besides a proof that the halting problem for Turing machines is not decidable by Turing machines they also verify a universal μ-recursive function, resulting in a universal Turing machine by translation.

[239] Xu et al.. 2013. Mechanising Turing machines and computability theory in Isabelle/HOL.

[4] Turing did not consider the halting problem. The definition of the Turing machine halting problem is due to Kleene [134], and the terminology "halting problem" due to Davis [47]. See the chapter by Copeland [37, pp. 39-45] and the paper by Lucas [159] for a precise historical account.

Asperti and Ricciotti mechanise single-tape [4] and multi-tape Turing machines [5] in the proof assistant Matita. They verify a translation between the two variants of Turing machines and a universal multi-tape Turing machine.

Norrish [178] formalises the full λ-caclulus in HOL4. He verifies a universal term and proves Rice's theorem. Forster and Smolka [83] formalise the weak call-by-value λ-calculus in Coq. They give a universal term, prove that total λ-definable relations yield total Coq functions, and prove Rice's theorem, while also analysing the constructive status of basic results in computability theory. Catt and Norrish [27] build on Norrish's work and prove the undecidability of Kolmogorov complexity.

Zammit [241] formalises μ-recursive functions in Coq and proves the S_n^m theorem. Pous [193] mechanises an equivalence proof between counter machines and μ-recursive functions in Coq. Larchey-Wendling [150] proves in Coq that total recursive functions can always be turned into Coq functions. Carneiro [26] formalises μ-recursive functions in Lean, gives a universal function, and proves Rice's theorem.

Ferreira Ramos, Ayala-Rincón, et al. [196, 67] formalise the partial recursive subset PVS0 of the PVS proof assistant, show that the respective halting problem is undecidable and prove Rice's theorem.

[4] Asperti and Ricciotti. 2012. Formalizing Turing Machines.

[5] Asperti and Ricciotti. 2015. A formalization of multi-tape Turing machines.

[178] Norrish. 2011. Mechanised Computability Theory.

[83] Forster and Smolka. 2017. Weak Call-by-Value Lambda Calculus as a Model of Computation in Coq.

[27] Catt and Norrish. 2021. On the formalisation of Kolmogorov complexity.

[241] Zammit. 1997. A Proof of the S-m-n theorem in Coq.

[193] Pous. 2004. A certified compiler from recursive functions to Minski machines.

[150] Larchey-Wendling. 2017. Typing total recursive functions in Coq.

[26] Carneiro. 2019. Formalizing Computability Theory via Partial Recursive Functions.

[196] Ramos et al.. 2018. Formalization of the Undecidability of the Halting Problem for a Functional Language.

[67] Ferreira Ramos et al.. 2020. Formalization of Rice's Theorem over a Functional Language Model.

10.3 Mechanisation in Coq

The code concerning this chapter is contributed or in the process of being contributed to the Coq Library of Undecidability Proofs. It can also be found on the following website:

`https://ps.uni-saarland.de/~forster/thesis`

In total, the Coq code concerned with Turing-equivalence proofs comprises 52k lines of code (LoC), with about 46% specifications and 54% proofs. Of them, around 12k LoC are library code contributed by various authors, 8k LoC belong to the certifying extraction framework for L and were jointly contributed by Fabian Kunze and the author of this thesis, and 14k LoC concerned with the compilation of binary stack machines to counter machines, the Diophantineness of FRACTRAN evaluation, and the μ-recursive solvability of Diophantine equations were contributed by Dominique Larchey-Wendling. Around 4k LoC verifying the Turing machine for the simulation of L were contributed mainly by Maximilian Wuttke and Fabian Kunze with some contributions by the author of this thesis. In total 14k LoC were contributed by the author of this thesis and Maximilian Wuttke as part of his Bachelor's thesis supervised by the author. We hyperlink the central theorems with the html version of the Coq code, indicated by a clickable 🐓-symbol.

For the present chapter, the key feature of Coq we rely on is setoid rewriting [214], i.e. rewriting with equivalence relations without specifying congruence lemmas manually. Forward reasoning able to deal with failure was also crucial for the verification of Turing machines, where we use the `smpl` plugin [206].

[214] Sozeau, Matthieu; Harvard University. 2009. A New Look at Generalized Rewriting in Type Theory.

For the present chapter, we conjecture that any other similarly expressive, tactic-based proof assistant with good support for rewriting with equivalence relations and some metaprogramming features would have worked similarly well. Our formalisation in constructive type theory makes explicit that the equivalence proofs are fully constructive. While some proofs use computational explosion, type casts, and guarded minimisation $\mu_{\mathbb{N}}$ for convenience, none of them seems integral, and all of them could likely be removed with some effort.

CHAPTER 11

The weak call-by-value λ-calculus L

The call-by-value λ-calculus was introduced by Plotkin [186] as variant of Church's λ-calculus [30]. The concrete variant of the call-by-value λ-calculus we present here is called L [83]. Programming in L is possible almost like in general-purpose functional programming languages from the ML family: Inductive types are supported by the use of Scott encodings and recursion is enabled by a recursion combinator. We discuss programming in L in Part IV, and here only define syntax, big-step semantics, and a stack machine with a heap for L.

[186] Plotkin. 1975. Call-by-name, call-by-value and the λ-calculus.

[30] Church. 1932. A set of postulates for the foundation of logic.

[83] Forster and Smolka. 2017. Weak Call-by-Value Lambda Calculus as a Model of Computation in Coq.

Publications The material in Section 11.1 is based on [84]. The stack machine semantics of 11.2 is based on [147, 77] and is mainly due to Fabian Kunze, we include it here to keep the thesis self-contained.

[84] Forster and Smolka. "Call-by-value lambda calculus as a model of computation in Coq." *Journal of Automated Reasoning* 63.2 (2019): 393-413.

[147] Kunze, Smolka, and Forster. "Formal small-step verification of a call-by-value λ-calculus machine." *Asian Symposium on Programming Languages and Systems*. 2018.

[77] Forster, Kunze, and Roth. "The weak call-by-value λ-calculus is reasonable for both time and space." *Proceedings of the ACM on Programming Languages 4*. POPL (2020).

11.1 Definition

We define the syntax of L using de Bruijn indices as the syntax of the full λ-calculus, i.e. as variables, applications, or abstractions.

$$s, t, u : \mathsf{tm}_{\mathsf{L}} ::= n \mid st \mid \lambda s \quad \textit{where } n : \mathbb{N}$$

We use names for concrete terms on paper, e.g. write $(\lambda xy.xx)(\lambda z.z)$ for $(\lambda\lambda 11)(\lambda 0)$.

We define a simple substitution operation s_t^n, agreeing with a more standard parallel substitution operation when the term substituted with is closed.

$$n_u^m := \textbf{if } n = m \textbf{ then } u \textbf{ else } n \qquad (st)_u^n := s_u^n t_u^n \qquad (\lambda s)_u^n := \lambda(s_u^{\mathsf{S}\,n})$$

Formally, we say that a term s is a **closed term** if $\forall nu.\ s_u^n = s$.

Def. closed term

We define big-step weak call-by-value evaluation $s \triangleright t$ following Plotkin [186].

$$\frac{}{\lambda s \triangleright \lambda s} \qquad \frac{s \triangleright \lambda u \qquad t \triangleright t' \qquad u_{t'}^0 \triangleright v}{st \triangleright v}$$

Note that we have for example $(\lambda xy.xx)(\lambda z.z) \triangleright \lambda y.(\lambda z.z)(\lambda z.z)$. Evaluation is called weak because the bodies of abstractions are not evaluated and call-by-value because arguments are evaluated before a function is called.

The L **halting problem** is defined as

Def. L halting problem

$$\mathsf{Halt}_{\mathsf{L}}(s : \mathsf{tm}_{\mathsf{L}}) := \exists t.\, s \triangleright t.$$

Obviously, we have $\mathsf{Halt}_{\mathsf{L}}((\lambda xy.xx)(\lambda z.z))$, but $\neg\mathsf{Halt}_{\mathsf{L}}((\lambda x.xx)(\lambda x.xx))$.
We immediately define a second halting problem for L for closed terms as

$$\mathsf{Halt}'_{\mathsf{L}}(x : \Sigma s : \mathsf{tm}_{\mathsf{L}}.\, \mathsf{closed}\, s) := \exists t.\, \pi_1 x \triangleright t.$$

Clearly, $\mathsf{Halt}'_{\mathsf{L}}\, x \preceq_m \mathsf{Halt}_{\mathsf{L}}$ via π_1. We will show that $\mathsf{Halt}_{\mathsf{L}} \preceq_m \mathsf{Halt}'_{\mathsf{L}}$ in part III using a universal machine, but already use the result in this part.

To define L-computability we define the Scott encoding of natural numbers:

$$\overline{0} := \lambda xy.x \qquad\qquad \overline{\mathsf{S}\, n} := \lambda xy.y\overline{n}$$

We will introduce recursive functions on natural numbers in Section 27.2.

Def. L-computable

A relation $R : \mathbb{N}^k \to \mathbb{N} \to \mathbb{P}$ is **L-computable** if

$$\exists s : \mathsf{tm}_{\mathsf{L}}.\mathsf{closed}\, s \wedge \forall n_1 \dots n_k.(\forall m.\, R(n_1,\dots,n_k)m \leftrightarrow s\overline{n_1}\dots\overline{n_k} \triangleright \overline{m}) \wedge$$
$$\forall t.\, s\overline{n_1}\dots\overline{n_k} \triangleright t \to \exists m.\, t = \overline{m}.$$

11.2 Stack machine semantics

The big-step semantics allows for a compact definition, but is not ideal for implementations of L. To prepare for a simulation of L on Turing machines we introduce a stack machine for L, utilising references to a heap instead of substitution, similar to the heap machine by Kunze, Smolka, and Forster [147]. In contrast to the results there, we give a direct correctness proof instead of a step-wise refinement via several machines. Our machine is also similar to the heap machine by Forster, Kunze, and Roth [77], but more geared towards verification. It can be used to show that L is reasonable for time, i.e. that one can define a time measure on L such that Turing machines can simulate L with polynomial overhead, but we omit all complexity-theoretic arguments.

Instead of terms, we will work with programs $P, Q : \mathsf{Pro} := \mathbb{L}\,\mathsf{Com}$, which are lists of commands. Commands are reference, application, abstraction, or return tokens:

$$c : \mathsf{Com} ::= \mathsf{ref}\, n \mid \mathsf{app} \mid \mathsf{lam} \mid \mathsf{ret}$$

Def. compilation function

The **compilation function** $\gamma : \mathsf{Ter} \to \mathsf{Pro}$ compiles terms to programs:

$$\gamma n := [\mathsf{ref}\ n] \qquad \gamma(st) := \gamma s +\!\!+ \gamma t +\!\!+ [\mathsf{app}] \qquad \gamma(\lambda s) := \mathsf{lam} :: \gamma s +\!\!+ [\mathsf{ret}]$$

We have $\gamma((\lambda xy.xx)(\lambda z.z)) = [\mathsf{lam}; \mathsf{lam}; \mathsf{ref}\ 1; \mathsf{ref}\ 1; \mathsf{app}; \mathsf{ret}; \mathsf{ret}; \mathsf{lam}; \mathsf{ref}\ 0; \mathsf{ret}; \mathsf{app}]$.

Compiled abstractions start with the token lam and end with ret. We can thus define a function $\delta P : \mathbb{O}(\mathsf{Pro} \times \mathsf{Pro})$ that extracts the body of an abstraction by matching the tokens like parentheses. We define $\delta P := \delta_{0,[]} P$, where $\delta_{k,Q} P$ is an auxiliary function storing the number k of unmatched lam and the processed prefix Q:

$\mathbb{O}$ → Sec. A.1, Page 207

$$\delta_{k,Q}\,[\,] := \mathsf{None} \qquad \delta_{0,Q}(\mathsf{ret} :: P) := \mathsf{Some}(Q, P) \qquad \delta_{\mathsf{S}\,k,Q}(\mathsf{ret} :: P) := \delta_{k,Q+\!\!+[\mathsf{ret}]} P$$

$$\delta_{k,Q}(\mathsf{lam} :: P) := \delta_{\mathsf{S}\,k,Q+\!\!+[\mathsf{lam}]} P \qquad \delta_{k,Q}(c :: P) := \delta_{k,Q+\!\!+[c]} P \ \text{ if } c = \mathsf{ref}\ n \text{ or } \mathsf{app}$$

We reuse the example term $(\lambda xy.xx)(\lambda z.z)$ from above and have:

$$\begin{aligned}&\delta[\mathsf{lam};\mathsf{ref}\,1;\mathsf{ref}\,1;\mathsf{app};\mathsf{ret};\mathsf{ret};\mathsf{lam};\mathsf{ref}\,0;\mathsf{ret};\mathsf{app}]\\ =&\mathsf{Some}([\mathsf{lam};\mathsf{ref}\,1;\mathsf{ref}\,1;\mathsf{app};\mathsf{ret}],[\mathsf{lam};\mathsf{ref}\,0;\mathsf{ret};\mathsf{app}])\end{aligned}$$

Def. states of the heap machine
Def. control stack
Def. value stack
Def. closures
Def. heap

The **states of the heap machine** are tuples T,V,H. The **control stack** T and the **value stack** V are lists of **closures** $g : \mathsf{Clos} := \mathsf{Pro} \times \mathbb{N}$. A closure (P,a) denotes an open program, where the reference 0 in P has to be looked up at address $a : \mathbb{N}$ in the heap when evaluating.

The **heap** H is a linked list of heap entries $e : \mathsf{Entry} := \mathbb{O}(\mathsf{Clos} \times \mathbb{N})$, i.e. an entry is either empty, or contains the head of the list and the address of its tail. Given a heap H and an address a, $H[a] : \mathbb{O}\,\mathsf{Entr}$ denotes the a-th element of H. We define $H[a,n]$ to be the n-th entry on the heap starting at address a as follows:

$$H[a,n] := \textbf{if } H[a] \textbf{ is } \mathsf{Some}(\mathsf{Some}(g,b)) \textbf{ then if } n \textbf{ is } \mathsf{S}\,n \textbf{ then } H[b,n] \textbf{ else } \mathsf{Some}\,g \textbf{ else } \mathsf{None}$$

We can now define the small-step semantics of the stack machine for L:

$$(\mathsf{lam} :: P,a) :: T,V,H \succ (P',a) ::_{\mathsf{tc}} T,(Q,a) :: V,H \quad \textit{if } \delta P = \mathsf{Some}(Q,P')$$

$$(\mathsf{ref}\,n :: P,a) :: T,V,H \succ (P,a) ::_{\mathsf{tc}} T,g :: V,H \quad \textit{if } H[a,n] = \mathsf{Some}\,g$$

$$(\mathsf{app} :: P,a) :: T,g :: (Q,b) :: V,H \succ (Q,|H|) :: (P,a) ::_{\mathsf{tc}} T,V,H +\!\!+ [\mathsf{Some}(g,b)]$$

Here, $(P,a) ::_{\mathsf{tc}} T := \textbf{if } P \textbf{ is } [\,] \textbf{ then } T \textbf{ else } (a,P) :: T$.

In the abstraction rule, the machine parses the complete abstraction using δ and puts the body on the value stack. In principle :: instead of $::_{\mathsf{tc}}$ could be used to obtain a correct machine, however the time complexity of this machine is easier to verify using this optimising operation. We refrain discussing time complexity of the machine, which is due to Fabian Kunze.

Similarly, in the reference rule, the machine looks up the body of the abstraction corresponding to the variable n in the heap starting at address a and puts the result on the value stack.

In the application rule, the machine takes closures of the called function (b,Q) and its argument g from the value stack. The address b is bound to g in the heap, the entry being appended to H, thus obtaining address $|H|$. The machine continues evaluating the body Q, where the value for reference 0 can be looked up at address $|H|$, where it was just placed.

Given a closed term s, the initial state of the machine is $([(\gamma s,0)],[\,],[\,])$, i.e. we start with an empty value stack, an empty heap, and the closure $(\gamma s,0)$ on the task stack. In fact, for a closed term s, the address 0 can be replaced by an arbitrary address since there will be no free reference to be looked up.

For example, a run of the machine for $(\lambda xy.xx)(\lambda z.z)$ reads as follows, using a as start address instead of 0:

$$\begin{aligned}&[([\mathsf{lam};\mathsf{lam};\mathsf{ref}\,1;\mathsf{ref}\,1;\mathsf{app};\mathsf{ret};\mathsf{ret};\mathsf{lam};\mathsf{ref}\,0;\mathsf{ret};\mathsf{app}],a)],[\,],[\,]\\ \succ&[([\mathsf{lam};\mathsf{ref}\,0;\mathsf{ret};\mathsf{app}],a)],[([\mathsf{lam};\mathsf{ref}\,1;\mathsf{ref}\,1;\mathsf{app};\mathsf{ret}],a)],[\,]\\ \succ&[([\mathsf{app}],a)],[([\mathsf{ref}\,0],a),([\mathsf{lam};\mathsf{ref}\,1;\mathsf{ref}\,1;\mathsf{app};\mathsf{ret}],a)],[\,]\\ \succ&[([\mathsf{lam};\mathsf{ref}\,1;\mathsf{ref}\,1;\mathsf{app};\mathsf{ret}],0)],[\,],[\mathsf{Some}([\mathsf{ref}\,0],a),a)]\\ \succ&[[\,],[([\mathsf{ref}\,1;\mathsf{ref}\,1;\mathsf{app}],0)],[\mathsf{Some}([\mathsf{ref}\,0],a),a)]]\end{aligned}$$

To state the correctness of the stack machine we need to define an unfolding operation $\mathsf{unf}_H(P,a)$. We will use functional notation for unfolding on paper, but define it as a functional relation in Coq, since the algorithm is not structurally recursive, and not even terminating on cyclic heaps. The function $\mathsf{unf} : \mathsf{Pro} \to \mathsf{tm_L}$ unfolds programs from the value stack into a term by inversing γ. It adds λ to the result, since only the bodies of abstractions are saved on the value stack. The function $\mathsf{unf}_{H,a,k} : \mathsf{tm_L} \to \mathsf{tm_L}$ substitutes free variables $n \geq k$ in a term by $H[a, n-k]$. Finally, $\mathsf{unf}_H : \mathsf{Pro} \times \mathbb{N} \to \mathsf{tm_L}$ unfolds a result using the two previous functions.

$$\begin{aligned}
\mathsf{unf}P &:= \lambda t && (\textit{if } \gamma t = P)\\
\mathsf{unf}_{H,a,k} n &:= n && (\textit{if } n < k)\\
\mathsf{unf}_{H,a,k} n &:= \mathsf{unf}_{H,b,0}(\mathsf{unf}P) && (\textit{if } n \geq k \textit{ and } H[a, n-k] = \mathsf{Some}(P,b))\\
\mathsf{unf}_{H,a,k}(st) &:= (\mathsf{unf}_{H,a,k} s)(\mathsf{unf}_{H,a,k} s)\\
\mathsf{unf}_{H,a,k}(\lambda s) &:= \mathsf{unf}_{H,a,\mathsf{S}\,k} s\\
\mathsf{unf}_H(P,a) &:= \mathsf{unf}_{H,a,0}(\mathsf{unf}P)
\end{aligned}$$

Continuing the example from above, we have

$$\begin{aligned}
&\mathsf{unf}_{[\mathsf{Some}(([\mathsf{ref}\,0],a),a)]}([\mathsf{ref}\,1; \mathsf{ref}\,1; \mathsf{app}], 0)\\
=\,&\mathsf{unf}_{[\mathsf{Some}(([\mathsf{ref}\,0],a),a)],0,0}(\mathsf{unf}[\mathsf{ref}\,1; \mathsf{ref}\,1; \mathsf{app}])\\
=\,&\mathsf{unf}_{[\mathsf{Some}(([\mathsf{ref}\,0],a),a)],0,0}(\lambda 11)\\
=\,&\lambda \mathsf{unf}_{[\mathsf{Some}(([\mathsf{ref}\,0],a),a)],0,1}(11)\\
=\,&\lambda(\mathsf{unf}_{[\mathsf{Some}(([\mathsf{ref}\,0],a),a)],0,1} 1)\,(\mathsf{unf}_{[\mathsf{Some}(([\mathsf{ref}\,0],a),a)],0,1} 1)\\
=\,&\lambda(\mathsf{unf}_{[\mathsf{Some}(([\mathsf{ref}\,0],a),a)],0,0}(\mathsf{unf}[\mathsf{ref}\,0]))\,(\mathsf{unf}_{[\mathsf{Some}(([\mathsf{ref}\,0],a),a)],0,0}(\mathsf{unf}[\mathsf{ref}\,0]))\\
=\,&\lambda(\lambda 0)(\lambda 0) = \lambda y.(\lambda z.z)(\lambda z.z)
\end{aligned}$$

The final correctness theorem then reads:

Theorem 11.1. Let s be closed.

1. If $s \triangleright t$ then $([(\gamma s, 0)], [], []) \succ^* ([], [(P,a)], H)$ for some P and a such that $\mathsf{unf}_H(P,a) = t$.
2. If $\sigma_s \succ^* (T,V,H)$ and $\neg\exists\sigma.(T,V,H) \succ \sigma$, then $T = []$, $V = [(P,a)]$, and $s \triangleright t$ for some P, a, and t such that $\mathsf{unf}_H(P,a) = t$ is defined.

CHAPTER 12

Turing machines

Turing machines [229] are the most low-level model of computation we consider. They have an explicit representation of data as strings stored on tapes unbounded in both directions. The transition function of a Turing machine is a table.

[229] Turing. 1936. On computable numbers, with an application to the Entscheidungsproblem.

Turing machines are widely used in books on computability theory and are the standard model of computation for complexity theory. Despite their universal use, there is no consensus on how to formally define Turing machines. We define multi-tape Turing machines following Asperti and Ricciotti [4, 5]. We give an overview of a Turing machine verification framework, which allows giving and verifying algorithms in the style of a register-based while-language. We introduce a compiler to machines with binary alphabet, a compiler to single-tape machines, and a universal machine. We then explain how to simulate L on Turing machines, based on the stack machine for L.

[4] Asperti and Ricciotti. 2012. Formalizing Turing Machines.

[5] Asperti and Ricciotti. 2015. A formalization of multi-tape Turing machines.

Turing machines and L constitute the two most extreme data points on the spectrum of models from machines which are easy to define and simulate, but where it is hard to program in to machines which are harder to define, tedious to simulate, but which have good abstractions for programming. The reason we opt for a direct translation instead of going via intermediate models is rooted in time complexity analyses of the translation not covered in this thesis, which would have been impeded by intermediate models.

Publications Sections 12.1, 12.2, 12.4, and 12.5 are based on [79]. The other sections contain adapted pieces of text from [78], which were written solely by the author of this thesis.

[79] Forster, Kunze, and Wuttke. "Verified programming of Turing machines in Coq." *Proceedings of the 8th ACM SIGPLAN Conference on Certified Programs and Proofs.* 2020.

[78] Forster, Kunze, Smolka, and Wuttke. "A Mechanised Proof of the Time Invariance Thesis for the Weak Call-by-value λ-Calculus." *International Conference on Interactive Theorem Proving.* 2021.

12.1 Definition

We start by defining a tape over type Σ using four constructors:

$$\mathsf{tp}_\Sigma ::= \mathsf{niltp} \mid \mathsf{leftof}\ r\ rs \mid \mathsf{midtp}\ ls\ m\ rs \mid \mathsf{rightof}\ l\ ls \qquad \textit{where } m, l, r : \Sigma \textit{ and } ls, rs : \mathbb{L}\Sigma$$

Following Asperti and Ricciotti [5], the representation does not allow for blank symbols, instead a blank symbol has to be part of the alphabet Σ. The seeming redundancy allows for a unique representation of every tape and disposes of well-formedness predicates.

We define a type of moves Move, a function mv applying a move to a tape, a function wr writing to a tape, and a function curr obtaining the current symbol of a tape in Figure 12.1 We define **Turing machine**s $M : \mathsf{TM}^n_\Sigma$ where $n : \mathbb{N}$ and Σ is a finite and discrete type as dependent pairs $(Q, \delta, q_0, \mathsf{halt})$ where Q is a finite and discrete type, $\delta : Q \times (\mathbb{O}\Sigma)^n \to Q \times (\mathbb{O}\Sigma \times \mathsf{Move})^n$, $q_0 : Q$ is the starting state, and $\mathsf{halt} : Q \to \mathbb{B}$ indicates halting states. The definition of Turing machine evaluation $M(q,t) \triangleright (q', t')$ and the **Turing machine halting problem** $\mathsf{Halt}_{\mathsf{TM}}$ are defined in Figure 12.1. The evaluation relation is deterministic.

Def. Turing machine

Def. TM^n_Σ

Def. Turing machine halting problem

Def. TM-computable

A relation $R : \mathbb{N}^k \to \mathbb{N} \to \mathbb{P}$ is TM-**computable** if

$$\exists n : \mathbb{N}.\ \exists \Sigma.\ \exists \mathsf{s}\,\mathsf{bl} : \Sigma.\ \mathsf{s} \neq \mathsf{bl} \wedge \exists M : \mathsf{TM}^{1+k+n}_{\Sigma}.\forall n_1 \dots n_k.$$
$$(\forall m.\, R(n_1,\dots,n_k)\, m \to \exists q\, t.\, M(q_0, (\mathsf{niltp}, \overline{n_1}, \dots, \overline{n_k}, \mathsf{niltp}, \dots, \mathsf{niltp})) \triangleright (q,t) \wedge t[0] = \overline{m}) \wedge$$
$$\forall q\, t\, i.\, M(q_0, (\mathsf{niltp}, \overline{n_1}, \dots, \overline{n_k}, \mathsf{niltp}, \dots, \mathsf{niltp})) \triangleright^i (q,t) \to \exists m. t[0] = \overline{m}$$

with $\overline{n} := \mathsf{midtp}\ [\,]\,\mathsf{bl}\,[\underbrace{\mathsf{s},\dots,\mathsf{s}}_{n \text{ times}}]$.

$$\mathsf{Move} ::= \mathsf{L} \mid \mathsf{N} \mid \mathsf{R}$$

$$\mathsf{mv} : \mathsf{Move} \to \mathsf{tp} \to \mathsf{tp}$$

$$\mathsf{mv}\,\mathsf{L}\,(\mathsf{rightof}\,l\,ls) := \mathsf{midtp}\,ls\,l\,[\,] \qquad \mathsf{mv}\,\mathsf{R}\,(\mathsf{leftof}\,r\,rs) := \mathsf{midtp}\,[\,]\,r\,rs$$

$$\mathsf{mv}\,\mathsf{L}\,(\mathsf{midtp}\,[\,]\,m\,rs) := \mathsf{leftof}\,m\,rs \qquad \mathsf{mv}\,\mathsf{R}\,(\mathsf{midtp}\,ls\,a\,[\,]) := \mathsf{rightof}\,a\,ls$$

$$\mathsf{mv}\,\mathsf{L}\,(\mathsf{midtp}\,(l :: ls)\,a\,rs) := \mathsf{midtp}\,ls\,l\,(a :: rs) \qquad \mathsf{mv}\,\mathsf{R}\,(\mathsf{midtp}\,ls\,a\,(r :: rs)) := \mathsf{midtp}\,(a :: ls)\,r\,rs$$

$$\mathsf{mv}\,m\,t := t \quad \textit{in all other cases}$$

$$\mathsf{wr} : \mathbb{O}\Sigma \to \mathsf{tp} \to \mathsf{tp}$$

$$\mathsf{wr}\,\mathsf{None}\,t := t \qquad \mathsf{wr}\,(\mathsf{Some}\,a)\,\mathsf{niltp} := \mathsf{midtp}\,[\,]\,a\,[\,] \qquad \mathsf{wr}\,(\mathsf{Some}\,a)\,(\mathsf{midtp}\,ls\,b\,rs) := \mathsf{midtp}\,ls\,a\,rs$$

$$\mathsf{wr}\,(\mathsf{Some}\,a)\,(\mathsf{leftof}\,r\,rs) := \mathsf{midtp}\,[\,]\,a\,(r :: rs) \qquad \mathsf{wr}\,(\mathsf{Some}\,a)\,(\mathsf{rightof}\,l\,ls) := \mathsf{midtp}\,(l :: ls)\,a\,[\,]$$

$$\mathsf{curr} : \mathsf{tp} \to \mathbb{O}\Sigma$$

$$\mathsf{curr}(\mathsf{midtp}\,ls\,a\,rs) := \mathsf{Some}\,a \qquad \mathsf{curr}\,t := \mathsf{None} \quad \textit{otherwise}$$

$$\frac{\mathsf{halt}\,q = \mathsf{true}}{M(q,t) \triangleright (q,t)} \qquad \frac{\mathsf{halt}\,q = \mathsf{false} \qquad \delta(q, \mathsf{curr}\,t) = (q', a) \qquad M(q', \mathsf{map}_2(\lambda(c,m)t.\mathsf{mv}\,m\,(\mathsf{wr}\,c\,t))\,a\,t) \triangleright (q'', t')}{M(q,t) \triangleright (q'', t')}$$

$$\mathsf{Halt}_{\mathsf{TM}^n_\Sigma}(M : \mathsf{TM}^n_\Sigma, t : \mathsf{tp}^n_\Sigma) := \exists q'\, t'.\, M(q_0, t) \triangleright (q', t')$$
$$\mathsf{Halt}_{\mathsf{TM}}(n : \mathbb{N}, \Sigma, M : \mathsf{TM}^n_\Sigma, t : \mathsf{tp}^n_\Sigma) := \mathsf{Halt}_{\mathsf{TM}^n_\Sigma}(M, t)$$

Figure 12.1.: Definitions for Turing machines

12.2 Verified programming of Turing machines

As presented, Turing machines are not compositional: There is no canonical way how to execute a 5-tape machine over alphabet $\mathbb{B}$ after a 3-tape machine over $\mathbb{O}(\mathbb{B} \times \mathbb{B})$.

Def. labelled Turing machine

To allow for the composition of Turing machines and their verification, we first introduce labellings in order to abstract away from the state space. A **labelled Turing machine** over a type L, written $M : \mathsf{TM}^n_\Sigma(L)$, is a dependent pair (M', lab_M) of a machine $M' : \mathsf{TM}^n_\Sigma$ and a labelling function $\mathsf{lab}_M : Q_{M'} \to L$.

Def. realisation

To prove the soundness of machines, we introduce **realisation**. A Turing machine M :

$\mathsf{TM}^n_\Sigma(L)$ realises a relation $R : \mathsf{tp}^n_\Sigma \to (L \times \mathsf{tp}^n_\Sigma) \to \mathbb{P}$ if

$$M \vDash R := \forall t\ q\ t'.\ M(q_0, t) \triangleright (q, t') \to R\ t\ (\mathsf{lab}_M\ q, t')$$

Dually, we introduce **termination**. $M : \mathsf{TM}^n_\Sigma(L)$ terminates in $T : \mathsf{tp}^n_\Sigma \to \mathbb{P}$ if **Def.** termination

$$M \downarrow T := \forall t.\ T\ t \to \exists q\ t'.\ M(t) \triangleright (q, t').$$

We call a machine total if $M \downarrow \lambda t.\top$, i.e. if it holds on any tape.

Fact 12.1. The introduced predicates are (anti-)monotone:

1. If $M \vDash R'$ and $\forall t\ \ell\ t'.\ R'\ t\ (\ell, t') \to R\ t\ (\ell, t')$, then $M \vDash R$.
2. If $M \downarrow T'$ and $\forall t.\ T\ t \to T'\ t$, then $M \downarrow T$.

We will use the following total machines we call **primitive machines**: **Def.** primitive machines

$$\mathsf{Read} : \mathsf{TM}^1_\Sigma(\mathbb{O}(\Sigma)) \vDash \lambda\ t\ (\ell, t').\ \ell = \mathsf{curr}\ t[0] \wedge t = t' \qquad \mathsf{Write}\ s(\mathbb{1}) : \mathsf{TM}^1_\Sigma \qquad \vDash \lambda\ t\ t'.\ t'[0] = \mathsf{wr}\ s\ t[0]$$

$$\mathsf{Move}\ d(\mathbb{1}) : \mathsf{TM}^1_\Sigma \vDash \lambda\ t\ t'.\ t'[0] = \mathsf{mv}\ d\ t[0] \qquad \mathsf{Return}\ \ell : \mathsf{TM}^n_\Sigma(L) \quad \vDash \lambda\ t\ (\ell', t').\ t' = t \wedge \ell' = \ell$$

The last necessary tool now are **combinators** to compose machines. Given $M : \mathsf{TM}^n_\Sigma(L)$ and $f : L \to \mathsf{TM}^n_\Sigma(L')$ we introduce the combinator $\mathsf{Switch}\ M\ f : \mathsf{TM}^n_\Sigma(L')$, which executes M and depending on the label ℓ returned by M executes $f\,\ell$. **Def.** combinators

$$\frac{M \vDash R \qquad \forall(\ell : L).\ f\,\ell \vDash R'_\ell}{\mathsf{Switch}\ M\ f \vDash \lambda t_0\ (\ell', t').\ \exists t\ (\ell : L).\ R\ t_0\ (\ell, t) \wedge R'_\ell\ t\ (\ell', t')} \qquad \frac{M \downarrow T \qquad M \vDash R \qquad \forall(\ell : L).f\,\ell \downarrow T'_\ell}{\mathsf{Switch}\ M\ f \downarrow \lambda\ t.\ \ T\ t \wedge\ \forall\ \ell\ t'.\ R\ t\ (\ell, t') \to T'_\ell\ t'}$$

We define the **sequential combinator** $M_1; M_2 := \mathsf{Switch}\ M_1\ (\lambda\ \_.\ M_2)$ such that **Def.** sequential combinator

Lemma 12.2. The following hold:

$$\frac{M_1 : \mathsf{TM}^n_\Sigma(L_1) \vDash R_1 \qquad M_2 : \mathsf{TM}^n_\Sigma(L_2) \vDash R_2}{M_1; M_2 : \mathsf{TM}^n_\Sigma(L_2) \vDash R_1 \circ R_2} \qquad \frac{M_1 \vDash R_1 \qquad M_1 \downarrow T_1 \qquad M_2 \downarrow T_2}{M_1; M_2 \downarrow \lambda t.\ T_1\ t \wedge \forall t'\ \ell.\ R_1\ t\ (\ell, t') \to T_2\ t'}$$

Given $M\_ : L_1 \to \mathsf{TM}^n_\Sigma(L_1 + L_2)$ and $\ell_0 : L_1$ we introduce the combinator $\mathsf{MemWhile}\ M\ \ell_0 : \mathsf{TM}^n_\Sigma(L_2)$, which first runs M_{ℓ_0}. If this results in label $\mathsf{inl}\,\ell_1$, MemWhile runs M_{ℓ_1}, and so on. Once M_{ℓ_n} results in $\mathsf{inr}\,\ell$, MemWhile returns ℓ. The realisation relation for MemWhile is defined inductively, whereas the termination relation is the co-inductively defined accessibility relation (the dashed line indicates coinduction).

$$\frac{\forall \ell : L_1.\ M_\ell \vDash R'_\ell}{\mathsf{MemWhile}\ M\ \ell_0 \vDash \mathsf{MemWhileR}\ R'\ \ell_0} \qquad \frac{M_{\ell_0} \vDash R_{\ell_0} \qquad \forall \ell.\ M_\ell \downarrow T_\ell}{\mathsf{MemWhile}\ M_{\ell_0} \downarrow \mathsf{MemWhileT}\ R\ T\ \ell_0}$$

$$\frac{R_{\ell_0}\ t\ (\mathsf{inl}\,\ell_1, t') \qquad \mathsf{MemWhileR}\ M\ \ell_1\ t'\ (\ell, t'')}{\mathsf{MemWhileR}\ M\ \ell_0\ t\ (\ell, t'')} \qquad \frac{R_{\ell_0}\ t\ (\mathsf{inr}\,\ell_2, t')}{\mathsf{MemWhileR}\ M\ \ell_0\ t\ (\ell_2, t')}$$

$$\frac{T_{\ell_0}\ t \qquad \forall t'\ell_1.R_{\ell_0}\ t\ (\mathsf{inl}\,\ell_1, t') \to \mathsf{MemWhileT}\ M\ \ell_1\ T\ R\ t'}{\mathsf{MemWhileT}\ M\ \ell_0\ T\ R\ t}$$

The composition operator from Lemma 12.2 can only compose machines over the same alphabet and the same number of tapes. To remedy this situation we introduce lifting operations for alphabets and tapes, and also a relabelling operation Relabel.

Given a retraction $f : \Sigma \to \Gamma$, a default symbol $d : \Sigma$, and a machine $M : \mathsf{TM}^n_\Sigma$, the **alphabet lift** $\Uparrow_{(f,d)} M : \mathsf{TM}^n_\Gamma$ translates every read symbol via f^{-1}, passes it to M, and translates the symbol M writes via f. In case f^{-1} returns None, d is passed to M.

Given a retraction $I : \mathbb{F}_m \to \mathbb{F}_n$ and a machine $M : \mathsf{TM}^m_\Sigma$, the **tape lift** $\Uparrow_I M : \mathsf{TM}^n_\Sigma$ replicates the behavior of M on tape i on tape Ii, and leaves all other tapes untouched.

Given a function $r : L_1 \to L_2$ and a machine $M : \mathsf{TM}^n_\Sigma(L_1)$, $\mathsf{Relabel}\ M\ r : \mathsf{TM}^n_\Sigma(L_2)$ behaves like M, but returns label $f\,\ell$ where M returned ℓ.

We omit the correctness relations of the constructions, since they are similar to what we have seen before.

12.3 Binary Turing machines

We describe a compiler from arbitrary single-tape Turing machines $M : \mathsf{TM}^1_\Sigma$ to binary single-tape Turing machines of type $\mathsf{TM}^1_{\mathbb{B}}$. Given the compiler from multi-tape Turing machines to single-tape machines described in the next section, this actually suffices to compile arbitrary multi-tape machines to binary single-tape machines.

We fix a machine $M : \mathsf{TM}^1_\Sigma$ with states in Q and alphabet $\Sigma = \{c_1, \ldots, c_n\}$.

We define an encoding function $\varepsilon : \mathbb{L}\Sigma \to \mathbb{L}\mathbb{B}$ as

$$\varepsilon[\,] := [\,] \qquad \varepsilon(c_i :: l) := \mathsf{false} :: \varepsilon c_i +\!\!+ \mathsf{true} :: \varepsilon l \qquad \varepsilon c_i := \mathsf{false}^i\mathsf{true}^{n-i}$$

and extend it to tapes:

$$\begin{aligned}
\varepsilon\,\mathsf{niltp} &:= \mathsf{niltp}\\
\varepsilon(\mathsf{midtp}\,ls\,c_i\,rs) &:= \mathsf{midtp}\,(\mathsf{rev}\,(\varepsilon(\mathsf{rev}\,ls)))\,\mathsf{false}\,(\varepsilon\,c_i +\!\!+ \mathsf{true} :: \varepsilon\,rs)\\
\varepsilon\,(\mathsf{leftof}\,c_i\,rs) &:= \mathsf{leftof}\,\mathsf{false}\,(\varepsilon\,c_i +\!\!+ \mathsf{true} :: \varepsilon\,ls)\\
\varepsilon\,(\mathsf{rightof}\,c_i\,ls) &:= \mathsf{rightof}\,\mathsf{true}\,(\mathsf{rev}\,(\varepsilon\,c_i) +\!\!+ \mathsf{false} :: (\mathsf{rev}\,(\varepsilon(\mathsf{rev}\,ls))))
\end{aligned}$$

We define the one-tape machine as $\mathsf{MemWhile}\ \mathsf{Step}_M$, where $\mathsf{Step}_M : Q \to \mathsf{TM}^1_{\mathbb{B}}(Q+Q)$ is a total machine, comprised of several other total machines with the following realisation relations:

$$\mathsf{ReadB} : \mathsf{TM}^1_{\mathbb{B}}(\Sigma) \vDash \lambda\ t\ (\ell, t').\forall t_\Sigma : \mathsf{tp}_\Sigma.t = \varepsilon\, t_\Sigma \to t' = t \wedge \ell = \mathsf{curr}\, t_\Sigma$$

$$\mathsf{WriteB}\ (c : \mathbb{O}\Sigma) : \mathsf{TM}^1_{\mathbb{B}}(\mathbb{1}) \vDash \lambda\ t\ t'.\forall t_\Sigma : \mathsf{tp}_\Sigma.t = \varepsilon\, t_\Sigma \to t' = \varepsilon(\mathsf{wr}\, c\, t_\Sigma)$$

$$\mathsf{MoveB}\ (m : \mathsf{Move}) : \mathsf{TM}^1_{\mathbb{B}}(\mathbb{1}) \vDash \lambda\ t\ t'.\forall t_\Sigma : \mathsf{tp}_\Sigma.t = \varepsilon\, t_\Sigma \to t' = \varepsilon(\mathsf{mv}\, m\, t_\Sigma)$$

$$\begin{aligned}
\mathsf{Step}_M\, q := \mathsf{Switch}\,\mathsf{ReadB}\,(\lambda c_i.&\textbf{if}\ \mathsf{halt}\, q = \mathsf{true}\ \textbf{then}\ \mathsf{Return}(\mathsf{inr}\, q)\\
&\textbf{else let}\ (q', m, c') := \delta(q, c_i)\\
&\textbf{in}\ \mathsf{WriteB}\, c'\,;\ \mathsf{MoveB}\, m\,;\ \mathsf{Return}(\mathsf{inl}\, q'))
\end{aligned}$$

Theorem 12.3. Let $M \vDash R$, $M \downarrow T$, and $\mathsf{Sim}\ q := \mathsf{Relabel}\ (\mathsf{MemWhile}\ \mathsf{Step}_M)\ q\ \mathsf{lab}_M$.

1. $\mathsf{Sim}\ q_0 \vDash \lambda t\ (\ell, t').\ \forall t_\Sigma. t = \varepsilon t_\Sigma \to \exists t'_\Sigma. R\ t_\Sigma\ (\ell, t'_\Sigma) \wedge t' = \varepsilon t'_\Sigma$
2. $\mathsf{Sim}\ q_0 \downarrow \lambda t. \exists t_\Sigma.\ t = \varepsilon t_\Sigma \wedge T\ t_\Sigma$

Proof. An inductive relation R'_q s.t. $\mathsf{Sim}\ q \vDash R'_q$ holds can be derived by definition. One then proves $\mathsf{Sim}\ q \vDash \lambda t\,(q', t').\ \forall t_\Sigma.\ t = \varepsilon t_\Sigma \to \exists t'_\Sigma. M(q, t_\Sigma) \triangleright (q', t)$ using Fact 12.1 by induction on R'_q. The claim follows again by Fact 12.1. The termination proof is dual. ∎

Corollary 12.4. $\mathsf{Halt}_{\mathsf{TM}^1} \preceq_m \mathsf{Halt}_{\mathsf{TM}^1_{\mathbb{B}}}$

12.4 Single tape Turing machines

We give a high-level overview of a compiler from multi-tape Turing machines $M : \mathsf{TM}^n_\Sigma$ to single-tape machines TM^1_Γ. As described in [79] we follow Sipser [207] and encode the content of n tapes over Σ, each with length n_i and content $c_{n,1} \dots c_{n,n_n}$, on a single tape as $\#c_{1,1} \dots c_{1,n_1} \# \dots \# c_{n,1} \dots c_{n,n_n}\$$. To mark the current symbol on each tape and the end of tapes, we work with the alphabet Γ also containing some administrative symbols:

$$\Gamma ::= \# \mid \$ \mid \mathsf{START} \mid \mathsf{STOP} \mid \underline{B} \mid \underline{\overleftarrow{B}} \mid \underline{\overrightarrow{B}} \mid \overleftarrow{B} \mid \overrightarrow{B} \mid x \mid \underline{x} \qquad (x : \Sigma)$$

Underlining denotes current symbols and B is a boundary symbol. The symbols START and STOP are technically not necessary, but simplify the implementation.

Given a tape vector $ts : \mathsf{tp}^n_\Sigma$ we define its encoding $\varepsilon(tp) : \mathbb{L}\Gamma$ as

$$\begin{array}{llll}
\varepsilon\ [\,] & := [\$] & \varepsilon\ (t :: ts) & := \# :: \varepsilon(t) +\!\!+ \varepsilon(ts) \\
\varepsilon\ (\mathsf{niltape}) & := [\underline{B}] & \varepsilon\ (\mathsf{leftof}\ r\ rs) & := \underline{\overleftarrow{B}} :: r +\!\!+ rs +\!\!+ [\overrightarrow{B}] \\
\varepsilon\ (\mathsf{midtape}\ ls\ m\ rs) & := \overleftarrow{B} :: \mathsf{rev}\, ls +\!\!+ \underline{m} :: rs +\!\!+ [\overrightarrow{B}] & \varepsilon\ (\mathsf{rightof}\ l\ ls) & := \overleftarrow{B} :: \mathsf{rev}\, ls +\!\!+ [l; \underline{\overrightarrow{B}}]
\end{array}$$

Similar to the compilation to binary machines, we implement the final machine using MemWhile and a step machine Step_M. Implementing Step_M is however considerably harder than before. To read the vector of n current symbols, which still can be passed as label from a finite type via MemWhile, Step_M has to traverse the whole tape. Similarly, writing and moving requires global operations on the whole tape. We thus omit the concrete implementation here and only state the correctness theorem.

To do so, we define for $t : \mathsf{tp}_\Gamma$ and $ts : \mathsf{tp}^n_\Sigma$ the containment relation $t \simeq ts$ as

$$t \simeq ts := t = \mathsf{midtape}\ [\,]\ \mathsf{START}\ (\varepsilon(ts) +\!\!+ [\mathsf{STOP}])$$

Theorem 12.5. Let $M \vDash R$ and $M \vDash T$. Then the following hold:

1. $\mathsf{Relabel}\ (\mathsf{MemWhile}\ \mathsf{Step}_M)\ \mathsf{lab}_M \vDash \lambda t\ t'. \forall ts.\ t \simeq ts \to \exists ts'. R\ ts\ ts' \wedge t' \simeq ts'$
2. $\mathsf{Relabel}\ (\mathsf{MemWhile}\ \mathsf{Step}_M)\ \mathsf{lab}_M \downarrow \lambda t. \exists ts. t \simeq ts \wedge T\ ts$

Corollary 12.6. $\mathsf{Halt}_{\mathsf{TM}} \preceq_m \mathsf{Halt}_{\mathsf{TM}^1}$

The proofs in Coq also reason about the time and space usage of the machines, which both are in $\mathcal{O}(n)$ of the usage of the initial machine. We omit these parts here, more details can be found in [79].

Note that we do not introduce TM^1-computability and go directly from TM-computability to BSM-computability.

12.5 A universal Turing machine

Universal machines are prevalent in computability theory. We here show how to implement and verify a universal Turing machine $\mathsf{Univ} : \mathsf{TM}^6_\Gamma$. Univ simulates the computation of single-tape machines $M : \mathsf{TM}^1_\Sigma$ on a tape $t_M : \mathsf{tp}_\Sigma$

For this verification, we introduce a new layer of abstraction treating tapes as registers, based on a notion $t[i] \simeq v$ expressing that tape $t[i]$ contains an encoded value $v : V$. A type V is a TM**-encodable type** on alphabet Σ if there is an injective function $\varepsilon : V \to \mathbb{L}\Sigma$.

Def. TM-encodable type

We define **tape containment** $t \simeq_f v$ where V is TM-encodable on alphabet Σ, $v : V$, $f : \Sigma \to \Gamma$, and $t : \mathsf{tp}_{\Gamma^+}$ is an injection as follows:

Def. tape containment

$$\Gamma^+ ::= \mathsf{START} \mid \mathsf{STOP} \mid \mathsf{UNKNOWN} \mid (s : \Gamma)$$

$$t \simeq_f v := \exists ls.\, t = \mathsf{midtp}\, ls\, \mathsf{START}\, (\mathsf{map}\, f\, (\varepsilon v) +\!\!+ [\mathsf{STOP}])$$

In Coq, we automatically infer f via type classes, and also omit f when specifying machines on paper. **Void** tapes (written $\mathsf{isvoid}\, t$) do not contain values. The head of the tape is located at the right-most symbol:

$$\mathsf{isvoid}\, t := \exists m\, ls.\ t = \mathsf{midtp}\, ls\, m\, [\,]$$

We omit the condition $\mathsf{isvoid}\, t$ when specifying machines and treat any unspecified tape as void.

In the Coq code, all notions are strengthened to allow the verification of the space consumption of machines, which we omit here.

Fact 12.7. The types $\mathbb{1}$, $\mathbb{B}$, and $\mathbb{N}$ are TM-encodable. If X and Y are encodable, then $\mathbb{O}(X)$, $\mathbb{L}(X)$, $X + Y$, and $X \times Y$ are encodable.

The central part of Univ is the encoding of δ_M. This is handled using a so-called association list, i.e. a list $\mathbb{L}(A \times B)$ where

$$A := \mathbb{O}(\Sigma_M) \times (\mathbb{B} \times \mathbb{N}) \qquad\qquad B := (\mathbb{O}(\Sigma_M) \times \mathsf{Move}) \times (\mathbb{B} \times \mathbb{N})$$

Note that $\mathbb{N}$ is used to encode states, and $\mathbb{B}$ is a flag indicating whether the state is halting. We implement a 5-tape machine Lookup which can look up the symbol to write, the move, and the successor state given the current symbol and current state.

The correctness theorem then reads:

Theorem 12.8. There is a machine $\mathsf{Univ} : \mathsf{TM}^6_\Gamma$ such that the following hold:

1. $\mathsf{Univ} \vDash \lambda t\, t'.\ \forall M\, q_0. \to t[1] \simeq \delta_M \to t[2] \simeq q_0 \to$
   $\forall q'\, t'_M.\ M(q_0, t[0]) \triangleright (q', t'_M) \to t'[0] = t'_M \wedge t'[1] \simeq \delta_M \wedge t'[2] \simeq q'$
2. $\mathsf{Univ} \vDash \lambda t.\ \exists M\, t'_M\, q'. t[1] \simeq \delta_M \wedge t[2] \simeq q' \wedge M(q_0, t[0]) \triangleright (q', t'_M)$

Using the compilers from the last two sections we can compile Univ to a class of binary single-tape universal Turing machines simulating single-tape machines over arbitrary alphabet

(parametrised over an alphabet Σ_M). Furthermore, if we see the compiler from multi-tape to single-tape machines as an encoding for n-tape machines, we can even define a class of binary single-tape single-tape universal Turing machines (again parametrised over Σ_M) simulating multi-tape Turing machines.

12.6 Simulating L on Turing machines

The simulation of L on Turing machines needs no new concepts. We simply implement the relation $\succ$ of the heap machine from Section 11.2 as multi-tape Turing machine $\mathsf{Step} : \mathsf{TM}^{11}_{\Sigma}$. The alphabet Σ consists of 30 symbols, allowing to encode commands, programs, addresses, closures, the task and value stack, heap entries, and the heap.

The two central components of Step are machines implementing the heap lookup operation $H[a,n]$ and the parsing operation δ:

Fact 12.9. There is a machine $\mathsf{Lookup} : \mathsf{TM}^5_{\Sigma}(\mathbb{B})$ such that the following hold:

1. $\mathsf{Lookup} \vDash \lambda t(\ell,t').\ \forall H\,a\,n.\ t[0] \simeq H \to t[1] \simeq a \to t[2] \simeq n \to$
   $\quad$ **if** ℓ **then** $\exists g.\ H[a,b] = \mathsf{Some}\,g \wedge t'[0] \simeq H \wedge t'[3] \simeq g$ **else** $H[a,b] = \mathsf{None}$
2. $\mathsf{Lookup} \downarrow \lambda t.\ \exists H\,a\,n.\ t[0] \simeq H \wedge t[1] \simeq a \wedge t[2] \simeq n$

Fact 12.10. There is a machine $\mathsf{Parse} : \mathsf{TM}^5_{\Sigma}(\mathbb{B})$ such that the following hold:

1. $\mathsf{Parse} \vDash \lambda t\,(\ell,t').\forall P.t[0] \simeq P \to$
   $\quad$ **if** ℓ **then** $\exists Q\,P'.\ \delta P = (Q,P') \wedge t'[o] \simeq P' \wedge t'[1] \simeq Q$ **else** $\delta P = \mathsf{None}$
2. $\mathsf{Parse} \downarrow \lambda t.\ \exists P.\ t[0] \simeq P$

While tedious in detail, in principle it is straightforward to compose the machine Step:

Fact 12.11. There is a machine $\mathsf{Step} : \mathsf{TM}^{11}_{\Sigma}(\mathbb{B})$ such that the following hold:

1. $\mathsf{Step} \vDash \lambda t\,(\ell,t').\ \forall T\,V\,H.\ t[0] \simeq T \to t[1] \simeq V \to t[2] \simeq H \to$
   $\quad$ **if** ℓ **then** $\exists T'\,V'\,H'.\ (T,V,H) \succ (T',V',H') \wedge t'[0] \simeq T' \wedge t'[1] \simeq V' \wedge t'[2] \simeq H'$
   $\quad\quad$ **else** $(\neg\exists\sigma.\ (T,V,H) \succ \sigma) \wedge T = [] \to t'[0] \simeq [] \wedge t'[1] \simeq V \wedge t'[2] \simeq H$
2. $\mathsf{Step} \vDash \lambda t.\ t[0] \simeq T \wedge t[1] \simeq V \wedge t[2] \simeq H$

Lemma 12.12. There are a machine $\mathsf{Sim} : \mathsf{TM}^{11}_{\Sigma}$ and a function init such that

$$(\exists T'\,V'\,H'.(T,V,H) \succ^* (T',V',H') \wedge \neg\exists\sigma.(T',V',H') \succ \sigma) \leftrightarrow \mathsf{Halt}_{\mathsf{TM}}(\mathsf{Sim}, \mathsf{init}(T,V,H)).$$

Proof. Sim can be defined as While Step. init initialises the tapes with the respective encodings of T, V, and H. ∎

Corollary 12.13. $\mathsf{Halt}_{\mathsf{L}} \preceq \mathsf{Halt}_{\mathsf{TM}}$

Corollary 12.14. Every L-computable relation $R : \mathbb{N}^k \to \mathbb{N} \to \mathbb{P}$ is TM-computable.

Proof. Given numbers $n_1,\dots,n_k$ on the first k tapes, the machine has to write the encoding of the term $s\,\overline{n}_1 \dots \overline{n}_k$ to a tape. Running Sim results in a heap containing the final result. Finally, running a machine implementing unf and a conversion from $\overline{m}$ to the TM-encoding of m yield the claim. ∎

12.7 Mechanisation

Turing machines are notoriously hard to mechanise, as already acknowledged by Asperti and Ricciotti [2012, 2015], Xu, Zhang, and Urban [239], and Ciaffaglione [32]. The literature contains various definitions of Turing machines. We almost literally follow Asperti and Ricciotti [5] for our definition. Here, we compare our definition to the one by Hopcroft, Motwani, and Ullman [113], chosen for instance by Wikipedia as reference definition [237]:

1. The logical system in [113] is classical set theory, whereas we work in constructive type theory. We discuss the impact of these foundational choices after the technical differences.
2. The alphabet in [113] is separated into a set of input symbols Σ and a superset of tape symbols Γ, where we unify both into a single type.
3. The blank symbol in [113] is an explicit part of Turing machines as an element of Γ, but not of Σ. We do not specify blank symbols explicitly and instead leave it to a user to specify a blank symbol or even various blank symbols.
4. Tapes in [113] are not formally defined. It is only stated that tapes "extend infinitely to the left and right, initially hold[ing] [...] the input". Instantaneous descriptions of Turing machines are formally defined as strings over Γ and Q, which contain "the portion of the tape between the leftmost and the rightmost nonblank, unless the head is to the left of the leftmost nonblank or to the right of the rightmost nonblank." In the latter case, the blanks between the head and the nonblank content are part of the string.
5. The transition function in [113] is a partial function, whereas ours is total. If the transition function is unspecified, the computation of the machine halts, whereas we have an explicit boolean halting function. In Coq's type theory one requires classical logic and $\mathsf{AUC}_{\mathbb{N},\mathbb{N}}$ to compile Turing machines with a partial transition function into an equivalent definition with total transition functions. In general, any compilation of partial functions on finite types to total functions is in contradiction to CT.
6. A machine in [113] always writes a symbol and always moves the head. We allow to not write a symbol and to not move the head. The first is important regarding our definition of tapes (otherwise a fully empty niltp can never stay fully empty), whereas the second is a relatively arbitrary choice to allow more freedom in the definition of concrete machines.
7. Turing machines have an explicit set of accepting states in [113]. Our more flexible definition of labels subsumes the binary notion of accepting states, but also allows for more interesting constructions like the Switch and MemWhile machines.

Subtle difficulties might arise when defining notions of computability theory in classical set theory. For instance, defining Turing machines with a transition function which takes as input the whole tape is problematic: arbitrary problems become decidable by encoding the decision into the transition function. When imposing the transition function to be computable one obtains a circular dependency: The notion of computability is needed to define transition functions, transition functions are needed to define Turing machines, but Turing machines are needed to define the notion of computability.

The well-known solution here is to define Turing machines as finite objects, i.e. use a transition table modelled by a (partial) function with finite domain and codomain. In classical set theory one can then always show afterwards that this transition function is computable, since every function with finite domain and codomain is.

In our type-theoretic setting, we can also show that the transition function is computable, provided it is, as we defined it, a total function. We will do so in Part III. Note however that the same would not be possible when employing a functional and total transition relation.

CHAPTER 13

Binary stack machines

Def. binary stack machines

We introduce **binary stack machines** BSMs as a device to simulate Turing machines. Essentially, binary stack machines can be seen as a variant of Turing machines, with three changes: (1) there is an explicit representation of programs as code consisting of instructions, (2) the state is a single natural number, the program counter, (3) the alphabet is fixed to $\mathbb{B}$, and (4) stacks are only unbounded in one dimension. Aspects (1)–(3) are also present in Post's definition of Turing machines [187]. The concrete formal definition of BSMs we use is due to Dominique Larchey-Wendling [81].

We give a high-level overview how to verify BSMs and define simple binary Turing machines SBTMs as intermediate model for the translation from $\mathsf{TM}^1_{\mathbb{B}}$ to BSMs. Translating from $\mathsf{TM}^1_{\mathbb{B}}$ to SBTMs abstracts away from the state space and uses $\mathbb{F}_n$ instead, ensures that there only is a single halting state and that the starting state has index 0, and encodes a tape more explicitly as $\mathbb{LB} \times \mathbb{OB} \times \mathbb{LB}$. Translating from SBTMs to BSMs involves expressing a tape as three counters, and expressing the expression table as sequence of instructions.

Publications The definition of SBTM and the reduction of TMs to SBTMs and further on to BSMs are novel. The definition of binary stack machines and the verification framework are due to Dominique Larchey-Wendling and are based on

[81] Forster and Larchey-Wendling. "Certified Undecidability of Intuitionistic Linear Logic via Binary Stack Machines and Minsky Machines." *Proceedings of the 7th ACM SIGPLAN Conference on Certified Programs and Proofs.* 2019.

13.1 Definition

For n-stack machines, a program (i, P) is a start label i and a list of instructions $P : \mathbb{L}\,\mathsf{instr}_n$. A state (c, v) is a program counter $c : \mathbb{N}$ and the stack configuration $v : (\mathbb{LB})^n$. We define

$$\mathsf{instr}_n ::= \mathsf{POP}\ (j : \mathbb{F}_n)\ (c_1 c_2 : \mathbb{N}) \mid \mathsf{PUSH}\ (j : \mathbb{F}_n)\ (b : \mathbb{B}).$$

The $\mathsf{POP}\ j\ c_1\ c_2$ instruction pops the top element the j-th stack. If this is true, the program counter is increased. If it is false, the program counter is set to c_1. If the stack was empty, the program counter is set to c_2. The $\mathsf{PUSH}\ j\ b$ instruction pushes b on top of the j-th stack and increases the program counter. We define the (deterministic) evaluation relation in Figure 13.1 and the BSM **halting problem** as

Def. BSM halting problem

$$\mathsf{Halt}_{\mathsf{BSM}}\,(n : \mathbb{N}, i : \mathbb{N}, P : \mathbb{L}\,\mathsf{instr}_n, v : (\mathbb{LB})^n) := \exists c'\, v'.\ (i, P)/\!/(c, v) \triangleright (c', v').$$

Def. BSM-computable

A relation $R : \mathbb{N}^k \to \mathbb{N} \to \mathbb{P}$ is BSM-**computable** if

$$\exists n. \exists i\,(P : \mathbb{L}\,\mathsf{instr}_{1+k+n}).\ \forall n_1 \ldots n_k.$$

$$(\forall m.\ R(n_1, \ldots, n_k)\, m \leftrightarrow$$

$$\exists c v. (i, P)/\!/(i, [[], \underbrace{[\mathsf{true}, \ldots, \mathsf{true}]}_{n_1 \text{ times}}, \ldots, \underbrace{[\mathsf{true}, \ldots, \mathsf{true}]}_{n_k \text{ times}}, [], \ldots, []]) \triangleright (c, \underbrace{[\mathsf{true}, \ldots, \mathsf{true}]}_{m \text{ times}} :: v)) \wedge$$

$$\forall c\, v.\ (i, P)/\!/(i, [[], \underbrace{[\mathsf{true}, \ldots, \mathsf{true}]}_{n_1 \text{ times}}, \ldots, \underbrace{[\mathsf{true}, \ldots, \mathsf{true}]}_{n_k \text{ times}}, [], \ldots, []]) \triangleright (c, v) \to \exists m\, v'.\ v = \underbrace{[\mathsf{true}, \ldots, \mathsf{true}]}_{m \text{ times}} :: v'.$$

$$\frac{c < i \vee i + |P| \leq c}{(i,P)/\!/(c,v) \triangleright (c,v)} \qquad \frac{c \geq i \quad P[c-i] = \mathsf{Some}(\mathsf{PUSH}\ j\ b) \quad (i,P)/\!/(c+1, v[j := b :: v[j]]) \triangleright (c',v')}{(i,P)/\!/(c,v) \triangleright (c',v')}$$

$$\frac{c \geq i \quad P[c-i] = \mathsf{Some}(\mathsf{POP}\ j\ c_1\ c_2) \quad v[j] = \mathsf{true} :: l \quad (i,P)/\!/(c+1, v[j := l]) \triangleright (c',v')}{(i,P)/\!/(c,v) \triangleright (c',v')} \qquad \frac{c \geq i \quad P[c-i] = \mathsf{Some}(\mathsf{POP}\ j\ c_1\ c_2) \quad v[j] = \mathsf{false} :: l \quad (i,P)/\!/(c_1, v[j := l]) \triangleright (c',v')}{(i,P)/\!/(c,v) \triangleright (c',v')}$$

$$\frac{c \geq i \quad P[c-i] = \mathsf{Some}(\mathsf{POP}\ j\ c_1\ c_2) \quad v[j] = [] \quad (i,P)/\!/(c_2, v) \triangleright (c',v')}{(i,P)/\!/(c,v) \triangleright (c',v')}$$

Figure 13.1.: Evaluation relation of binary stack machines

13.2 Verified programming of binary stack machines

We briefly describe how to verify binary stack machines compositionally. For BSMs, compositionality is based on a notion of **subprograms**. We call (i_1, P_1) a subprogram of (i_2, P_2) if $\exists l\, r. P_2 = l +\!\!+ P_1 +\!\!+ r \wedge i_1 = i_2 + |l|$. Furthermore, a program counter c is considered **out of code** for a program (i, P) if $c < i \vee c \geq |P| + i$.

Def. subprograms

Def. out of code

The program counter of any result of an evaluation is out of code (otherwise the computation would continue):

Fact 13.1. If $(i,P)/\!/(c,v) \triangleright (c',v')$ then c' is out of code for (i,P).

Properties of programs can be verified by compositionally verifying the computation of subprograms:

Fact 13.2. If (i_1,P_1) is a subprogram of (i_2,P_2), $(i_1,P_1)/\!/(c,v) \triangleright (c',v')$, and $(i_2,P_2)/\!/(c',v') \triangleright (c'',v'')$, then $(i_2,P_2)/\!/(c,v) \triangleright (c'',v'')$.

Dually, the evaluation of a program can be inverted into evaluation of its subprograms:

Fact 13.3. If (i_1,P_1) is a subprogram of (i_2,P_2), $(i_2,P_2)/\!/(c,v) \triangleright (c',v')$, and $(i_1,P_1)/\!/(c,v) \triangleright (c'',v'')$, then $(i_2,P_2)/\!/(c'',v'') \triangleright (c',v')$.

13.3 Simple binary Turing machines

Simple binary Turing machines (SBTM) simplify five aspects in comparison to $\mathsf{TM}^1_{\mathbb{B}}$. First, they use tapes rather than unary vectors of tapes. While on paper we identify the two types, in the Coq mechanisation this difference matters considerably. Secondly, they work with a simplified representation of tapes, namely $\mathbb{LB} \times \mathbb{OB} \times \mathbb{LB}$. Thirdly, SBTMs do not work with a finite type of states, but use $\mathbb{F}_{\mathsf{S}n}$ for some n as type of states. Fourthly, they do not have a halting function and halting states are identified by returning None in the transition function. Lastly, there is no dedicated starting state, instead $0 : \mathbb{F}_{\mathsf{S}n}$ is always the starting state. We define

$$\mathsf{SBTM} := \Sigma n : \mathbb{N}.\ (\mathbb{F}_{\mathsf{S}n} \times \mathbb{OB}) \to \mathbb{O}(\mathbb{F}_{\mathsf{S}n} \times \mathbb{OB} \times \mathsf{move}).$$

Σ → Sec. A.1, Page 207

We write δ_M for the second component of a machine, and $|M|$ for the first. The evaluation

relation on configurations of type $\mathbb{F}_{\mathsf{S}(|M|)} \times (\mathbb{LB} \times \mathbb{OB} \times \mathbb{LB})$ is defined as

$$\frac{\delta_M(q,c) = \mathsf{None}}{M(q,(ls,o,rs)) \triangleright (q,(ls,o,rs))} \qquad \frac{\delta_M(q,c) = \mathsf{Some}\,(q',w,m) \quad M(q',\mathsf{mv}\ m\ (\mathsf{wr}\ w\ (ls,o,rs))) \triangleright (q'',t')}{M(q,(ls,o,rs)) \triangleright (q'',t')}$$

where

$$\begin{aligned}
\mathsf{wr}\ (\mathsf{Some}\,c)\ (ls,o,rs) &:= (ls,\mathsf{Some}\,c,rs) & \mathsf{wr}\ \mathsf{None}\ t &:= t\\
\mathsf{mv\ L}\ (l :: ls,\mathsf{None},rs) &:= (ls,\mathsf{Some}\,l,rs) & \mathsf{mv\ L}\ (l :: ls,\mathsf{Some}\,c,rs) &:= (ls,\mathsf{Some}\,l,c :: rs)\\
\mathsf{mv\ L}\ ([],\mathsf{Some}\,c,rs) &:= ([],\mathsf{None},c :: rs) & \mathsf{mv\ R}\ (ls,\mathsf{None},r :: rs) &:= (ls,\mathsf{Some}\,r,rs)\\
\mathsf{mv\ R}\ (ls,\mathsf{Some}\,c,r :: rs) &:= (c :: ls,\mathsf{Some}\,r,rs) & \mathsf{mv\ R}\ (ls,\mathsf{Some}\,c,[]) &:= (c :: ls,\mathsf{None},[])\\
\mathsf{mv}\ m\ t &:= t
\end{aligned}$$

Note that in principle, SBTMs could be simplified further: Instead of returning $1 + 9 \cdot \mathsf{S}\,|M|$ different possible actions in δ, we could reduce to the actions "halt", "move left and set state to q'", "move right and set state to q'", and "write c and set state to q'", i.e. $1 + 3 \cdot \mathsf{S}\,|M|$ many. However, this would not fundamentally simplify proofs, and proof automation in the mechanisation seems to make the simplification superfluous.

The SBTM **halting problem** is defined as

Def. SBTM halting problem

$$\mathsf{Halt}_{\mathsf{SBTM}}(M,t) := \exists q'\,t'.M(0,t) \triangleright (q',t')$$

Theorem 13.4. $\mathsf{Halt}_{\mathsf{TM}} \preceq_m \mathsf{Halt}_{\mathsf{SBTM}}$

Proof. Translating unary vectors of tapes to triples is straightforward. Given a machine M, translating its states to $\mathbb{F}_{\mathsf{S}|M|}$ as $\mathsf{S}\,(f\,q)$ via the bijection from Corollary 3.25. If a state is halting, $\delta(\mathsf{S}\,(f\,q),o) := \mathsf{None}$. Otherwise, δ can be inferred directly from δ_M. Lastly, we define $\delta(0,o) := \mathsf{Some}\,(\mathsf{S}\,(f\,q),\mathsf{None},\mathsf{N})$, i.e. the starting state of the constructed SBTM simply transitions to the starting state of M. ∎

Note that we do not introduce SBTM-computability and go directly from TM-computability to BSM-computability.

13.4 Simulating SBTMs on BSMs

We simulate a simple binary Turing machine M : SBTM on a binary stack machine with 4 stacks. Three stacks are used to represent the left, current, and right part of the tape (we denote the stacks with LEFT, CURR, RIGHT), while the fourth stack (denoted EMPTY) is always empty and used to execute (unconditional) jumps. We define JMP c := POP EMPTY c c.
A tape is encoded as stack configuration via

$$\varepsilon(ls,o,rs) := (ls,\mathbf{if}\ o\ \mathbf{is}\ \mathsf{Some}\,c\ \mathbf{then}\ [c]\ \mathbf{else}\ [],rs,[]).$$

It is relatively straightforward to implement a machine MOVE moving the tape:

Lemma 13.5. Given $q : \mathbb{F}_{S|M|}$, $o : \mathbb{OB}$, and $i : \mathbb{N}$ there is a machine MOVE q o i with $|\text{MOVE } q\ o\ i| = 23$ such that the following hold:

$$(i, \text{MOVE } q\ o\ i)/\!/(i, \varepsilon t) \triangleright (i+23, \varepsilon(\text{mv } m\ t)) \qquad (\text{if } \delta(q,o) = \text{Some}(q', w, m))$$

$$(i, \text{MOVE } q\ o\ i)/\!/(i, \varepsilon t) \triangleright (i+23, \varepsilon t) \qquad (\text{if } \delta(q,o) = \text{None})$$

Given MOVE we can define the program PROG_q corresponding to a state q as follows, where $|\text{PROG}_q| = 76$, off $:= 76 \cdot q$, and END $= (2+n) \cdot 76$:

$$\begin{aligned}
\text{off}: &\ \text{POP CURR } (26+\text{off})\ (51+\text{off})\\
1+\text{off}: &\ \text{PUSH CURR } (\textbf{if } \delta(q, \text{Some true}) \textbf{ is Some } (_, \text{Some false}, _) \textbf{ then false else true})\\
2+\text{off}: &\ \text{MOVE } i\ (\text{Some true})\ (2+\text{off})\\
25+\text{off}: &\ \text{JMP } (\textbf{if } \delta(q, \text{Some false}) \textbf{ is Some } (q', _, _) \textbf{ then } c \cdot q' \textbf{ else END})\\
26+\text{off}: &\ \text{PUSH CURR } (\textbf{if } \delta(q, \text{Some false}) \textbf{ is Some } (_, \text{Some true}, _) \textbf{ then true else false})\\
27+\text{off}: &\ \text{MOVE } i\ (\text{Some false})\ (27+\text{off})\\
50+\text{off}: &\ \text{JMP } (\textbf{if } \delta(q, \text{Some false}) \textbf{ is Some } (q', _, _) \textbf{ then } c \cdot q' \textbf{ else END})\\
51+\text{off}: &\ \textbf{match } \delta(q, \text{None}) \textbf{ with None} \Rightarrow \text{JMP END}\\
&\qquad\qquad \mid \text{Some } (_, \text{Some } w, _) \Rightarrow \text{PUSH CURR } w\\
&\qquad\qquad \mid \text{Some } _ \Rightarrow \text{JMP } (52+\text{off}) \textbf{ end}\\
52+\text{off}: &\ \text{MOVE } i\ \text{None}(52+\text{off})\\
75+\text{off}: &\ \text{JMP } (\textbf{if } \delta(q, \text{None}) \textbf{ is Some } (q', _, _) \textbf{ then } c \cdot q' \textbf{ else END})
\end{aligned}$$

Lemma 13.6. $(76 \cdot q, \text{PROG}_q)/\!/(c \cdot q, \varepsilon(ls, o, rs)) \triangleright$ **if** $\delta(i,o)$ **is** Some (q', w, m) **then** $(c \cdot q', \varepsilon(\text{mv } m\ (\text{wr } w\ t)))$ **else**$(\text{END}, \varepsilon(ls, o, rs))$

Theorem 13.7. Given a SBTM $M = (n, \delta)$ we have that the following hold:

1. If $M(t) \triangleright t'$ then $\exists c.\ (0, \text{PROG}_0 +\!\!+ \cdots +\!\!+ \text{PROG}_n)/\!/(0, \varepsilon t) \triangleright (c, \varepsilon t')$.
2. If $(0, \text{PROG}_0 +\!\!+ \cdots +\!\!+ \text{PROG}_n)/\!/(0, \varepsilon t) \triangleright (c, v)$ then $\exists t'.\ M(t) \triangleright t'$.

Corollary 13.8. $\text{Halt}_{\text{SBTM}} \preceq_m \text{Halt}_{\text{BSM}}$

Corollary 13.9. Every TM-computable relation $R: \mathbb{N}^k \to \mathbb{N} \to \mathbb{P}$ is BSM-computable.

13.5 Mechanisation

Binary stack machines are not a standard model of computation, so we refrain from comparing them to similar models found in the literature. Several formalisation choices were however taken that could be taken differently. For instance, there is no unconditional jump operation, meaning an unconditional jump needs an additional register. Thus, by some extra work, one can easily prove that the halting problem for 3-stack binary stack machines is undecidable (by encoding the content of CURR into the state space), but it is unclear what the situation for machines with 2 stacks is. The decision that the machine increases the program counter when popping true is arbitrary. This choice does however matter less for concrete results and is mainly a convention one has to keep in mind when programming.

The verification of the BSMs defined in Section 13.4 is not carried out using the shown evaluation relation, but based on the small step semantics used by Forster and Larchey-Wendling [81]. In particular, the general framework for the definition and verification of models of computation defined using small step semantics and programs due to Dominique Larchey-Wendling was very helpful. The main mathematical ideas based on small step semantics coincide with the ideas explained in Section 13.2.

CHAPTER 14

Counter machines

Counter machines (CMs) were independently introduced by various authors, see Appendix A of the paper by Shepherdson and Sturgis [205]. Counter machines have programs composed of instructions, a program counter, and a fixed amount of counters that can be manipulated by the current instruction. Depending on the author, the instruction sets of counter machines differ. We use n-counter machines with only increment and decrement instructions as defined by Minsky[170, Chapter 14]. Counter machines are oftentimes also called register machines, abacus machines, or after the person who first defined the particular flavour of counter machines.

[205] Shepherdson and Sturgis. 1963. Computability of Recursive Functions.

[170] Minsky. 1967. Computation: Finite and Infinite Machines.

To translate BSMs to CMs, we encode a binary stack as natural number and BSM instructions as sequence of CM instructions, meaning program indices for jumps have to be adjusted accordingly.

Publications The definition of counter machines is based on [81].[1] This paper also contains a general compiler for instruction-based models of computation. We do not explain the compiler, but only the operation of the compiler on the instances BSM and CM.

[81] Forster and Larchey-Wendling. "Certified Undecidability of Intuitionistic Linear Logic via Binary Stack Machines and Minsky Machines." *Proceedings of the 7th ACM SIGPLAN Conference on Certified Programs and Proofs*. 2019.

14.1 Definition

For n-counter machines, a program (i, P) is a start label i and a list of instructions $P : \mathbb{L}\,\mathsf{instr}_n$. A state (c, v) is a program counter $c : \mathbb{N}$ and the stack configuration $v : \mathbb{N}^n$.

Instructions for n-stack machines are defined as follows:

$$\mathsf{instr}_n ::= \mathsf{DEC}\ (j : \mathbb{F}_n)\ (c : \mathbb{N}) \mid \mathsf{INC}\ (j : \mathbb{F}_n)$$

The DEC j c instruction decreases the j-th counter and increases the program counter. If counter j is already 0, it is left at 0 and the program counter set to c. The INC j instruction increases counter j and the program counter. We define the (deterministic) evaluation relation in Figure 14.1 and the CM **halting problem** as

Def. CM halting problem

$$\mathsf{Halt}_{\mathsf{CM}}(n : \mathbb{N}, i : \mathbb{N}, P : \mathbb{L}\,\mathsf{instr}_n, v : \mathbb{N}^n) := \exists c'\, v'.\ (i, P)/\!/(i, v) \triangleright (c', v').$$

Def. $\mathsf{Halt}_{\mathsf{CM}}$

A relation $R : \mathbb{N}^k \to \mathbb{N} \to \mathbb{P}$ is CM-**computable** if

Def. CM-computable

$$\exists n. \exists i\, (P : \mathbb{L}\,\mathsf{instr}_{1+k+n}).\ \forall n_1 \ldots n_k.$$
$$(\forall m.\ R(n_1, \ldots, n_k)\, m \leftrightarrow \exists c v. (i, P)/\!/(i, [0, n_1, \ldots, n_k, 0, \ldots, 0]) \triangleright (c, m :: v)).$$

[1] Note that there counter machines are called Minsky machines (MM).

$$\frac{c < i \lor i + |P| \le c}{(i,P)/\!/(c,v) \triangleright (c,v)} \qquad \frac{c \ge i \qquad P[c-i] = \mathsf{Some}(\mathsf{INC}\ j) \qquad (i,P)/\!/(c+1, v[j := v[j]+1]) \triangleright (c',v')}{(i,P)/\!/(c,v) \triangleright (c',v')}$$

$$\frac{c \ge i \qquad P[c-i] = \mathsf{Some}(\mathsf{DEC}\ j\ c'') \qquad v[j] = \mathsf{S}\,n \qquad (i,P)/\!/(c+1, v[j := n]) \triangleright (c',v')}{(i,P)/\!/(c,v) \triangleright (c',v')}$$

$$\frac{c \ge i \qquad P[c-i] = \mathsf{Some}(\mathsf{DEC}\ j\ c'') \qquad v[j] = 0 \qquad (i,P)/\!/(c'',v) \triangleright (c',v')}{(i,P)/\!/(c,v) \triangleright (c',v')}$$

Figure 14.1.: Evaluation relation of counter machines

Since counter machines natively work on natural numbers, no second condition is needed.

14.2 Simulating BSMs on CMs

Let an n-stack binary stack machine (i,P) be given. We interpret a stack $[b_1,\dots,b_n]$ as the natural number corresponding to the binary number $b_n \cdots b_1$, i.e.

$$\varepsilon[\,] := 1 \qquad \varepsilon(\mathsf{true} :: l) := 1 + 2\cdot\varepsilon l \qquad \varepsilon(\mathsf{false} :: l) := 2\cdot\varepsilon l$$

To work with encoded stacks we need three auxiliary programs: A program adding two counters, a program multiplying the value of a counter by 2, and a program computing division and modulo 2 in one go.

Note that for all programs we will need a counter which is guaranteed to be 0 to implement unconditional jumps. Due to the definition of the instruction set, decreasing a counter until it is 0 is only possible if there already is a 0 counter.

Fact 14.1. Given a number of counters n, different counters $\mathsf{src}, \mathsf{dst}, \mathsf{tmp} : \mathbb{F}_n$, and an index i there are programs add, divtwo, and multwo such that if $v[\mathsf{tmp}] = 0$ we have:

$$(i,\mathsf{add})/\!/(i,v) \triangleright (3+i, v[\mathsf{src} := 0, \mathsf{dst} := k + x])$$

$$v[\mathsf{src}] = 2\cdot k \to (i,\mathsf{divtwo})/\!/(i,v) \triangleright (6+i, v[\mathsf{src} := 0, \mathsf{dst} := k + v[\mathsf{dst}], \mathsf{tmp} := 0])$$

$$v[\mathsf{src}] = 1 + 2\cdot k \to (i,\mathsf{divtwo})/\!/(i,v) \triangleright (6+i, v[\mathsf{src} := 0, \mathsf{dst} := k + v[\mathsf{dst}], \mathsf{tmp} := 1])$$

$$(i,\mathsf{multwo})/\!/(i,v) \triangleright (4+i, v[\mathsf{src} := 0, \mathsf{dst} := 2\cdot v[\mathsf{src}] + v[\mathsf{dst}]])$$

We simulate (i,P) by a counter machine (i,Q) with $2+n$ stacks and thus lift ε to a function $\varepsilon : (\mathbb{LB})^n \to \mathbb{N}^{2+n}$ by defining $\varepsilon v := 0 :: 0 :: \mathsf{map}\ \varepsilon\ v$. Every instruction in P is simulated by a subprogram of Q composed of the auxiliary programs defined before. We refer to the first and second counter of Q as tmp_1 and tmp_2. Before and after simulating an instruction, tmp_1 and tmp_2 will both contain 0.

Lemma 14.2. There are $2+n$-counter programs $\mathsf{POP}\ i\ j\ c_1\ c_2$, $\mathsf{PUSHT}\ i\ j$, $\mathsf{PUSHF}\ i\ j : \mathsf{instr}_{2+n}$ where $i : \mathbb{N}$, $j : \mathbb{F}_n$ and $c_1, c_2, c_3 : \mathbb{N}$ with length k_{POP}, k_{PUSHT}, and k_{PUSHT} respectively, such that for all $v : (\mathbb{LB})^n$:

$$v[j] = \mathsf{true} :: l \to (i, \mathsf{POP}\ i\ j\ c_1\ c_2\ c_3)/\!/(i,\varepsilon v) \triangleright (c_1, \varepsilon(v[j := l]))$$

$$v[j] = \mathsf{false} :: l \to (i, \mathsf{POP}\ i\ j\ c_1\ c_2\ c_3)/\!/(i,\varepsilon v) \triangleright (c_2, \varepsilon(v[j := l]))$$

$$v[j] = [] \to (i, \mathsf{POP}\ i\ j\ c_1\ c_2\ c_3) /\!/ (i, \varepsilon v) \triangleright (c_3, \varepsilon v)$$
$$(i, \mathsf{PUSHT}\ i\ j) /\!/ (i, \varepsilon v) \triangleright (7+i, \varepsilon(v[j := \mathsf{true} :: v[j]]))$$
$$(i, \mathsf{PUSHF}\ i\ j) /\!/ (i, \varepsilon v) \triangleright (7+i, \varepsilon(v[j := \mathsf{false} :: v[j]]))$$

Proof. We define $\mathsf{PUSHF}\ i\ j := \mathsf{add}\ i\ (2+j)\ \mathsf{tmp}_1\ \mathsf{tmp}_2 +\!\!+ \mathsf{multwo}\ i\ \mathsf{tmp}_1\ (2+j)\ \mathsf{tmp}_2$ and $\mathsf{PUSHT}\ i\ j := \mathsf{PUSHF}\ i\ j +\!\!+ [\mathsf{INC}\ (2+j)]$. The POP program is relatively complicated, we omit it here. ∎

Note that the programs in the last lemma are not all of the same length, but the length also does not depend on either of the parameters. Even before defining a compilation function γ, we can define a labelling function as follows:

$$\ell'[] := 1 \qquad \ell'(\mathsf{POP}\ j\ c_1\ c_2 :: P) := k_{\mathsf{POP}} + \ell' P$$
$$\ell'(\mathsf{PUSH}\ j\ \mathsf{true} :: P) := k_{\mathsf{PUSHT}} + \ell' P \qquad \ell'(\mathsf{PUSH}\ j\ \mathsf{false} :: P) := k_{\mathsf{PUSHF}} + \ell' P$$

$$\ell c := \textbf{if}\ i \le c < i + |P|\ \textbf{then}\ \ell'(\mathsf{take}\ c\ P)\ \textbf{else}\ 1 + \ell' P$$

Thus, label c in P will correspond to label ℓc in Q. We define $Q := \gamma_0 P$ where

$$\gamma_i(\mathsf{POP}\ j\ c_1\ c_2 :: P) := \mathsf{POP}\ (\ell i)\ (2+j)\ (\ell(1+i))\ (\ell c_1)\ (\ell c_2) +\!\!+ \gamma_{1+i} P$$
$$\gamma_i(\mathsf{PUSH}\ j\ \mathsf{true} :: P) := \mathsf{PUSHT}\ (\ell i)\ (2+j) +\!\!+ \gamma_{1+i} P$$
$$\gamma_i(\mathsf{PUSH}\ j\ \mathsf{false} :: P) := \mathsf{PUSHF}\ (\ell i)\ (2+j) +\!\!+ \gamma_{1+i} P$$

Fact 14.3. Let $i \le c < i + |P|$. Then $\ell c = |\gamma_0(\mathsf{take}\ c\ P)|$.

Theorem 14.4. The following hold:

1. $(i, P) /\!/_{\mathsf{BSM}} (i, v) \triangleright (c, v') \to (1, Q) /\!/_{\mathsf{CM}} (1, \varepsilon v) \triangleright (1 + |M|, \varepsilon v')$
2. $(1, Q) /\!/_{\mathsf{CM}} (1, \varepsilon v) \triangleright (c, w) \to \exists c' v'.\ (i, P) /\!/_{\mathsf{BSM}} (i, v) \triangleright (c, v')$

Corollary 14.5. $\mathsf{Halt}_{\mathsf{BSM}} \preceq_m \mathsf{Halt}_{\mathsf{CM}}$

Corollary 14.6. Every BSM-computable relation $R : \mathbb{N}^k \to \mathbb{N} \to \mathbb{P}$ is CM-computable.

CHAPTER 15

FRACTRAN

[35] Conway. 1987. Fractran: A simple universal programming language for arithmetic.

FRACTRAN was invented as an esoteric programming language by Conway [35]. FRACTRAN programs are lists of (positive) fractions, and a state is comprised of a single natural number. In state s, the program steps to $s \cdot f$, where f is the first fraction in the program such that $s \cdot f$ is a natural number. If no such f exists, the program halts. By interpreting the state $(c, [x_1, \ldots, x_n])$ of a counter machine as product of prime numbers, FRACTRAN can simulate CM programs, which we explain in Section 15.2.

Publications The mechanisation of FRACTRAN and the simulation of CM is from

[152] Larchey-Wendling and Forster. "Hilbert's Tenth Problem in Coq." *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*. 2019.

15.1 Definition

A FRACTRAN program is a list of fractions P, modelled as pairs of natural numbers p/q. The state of a FRACTRAN program is single natural number n. The one step relation $\succ$, and the evaluation relation $\triangleright$ are defined as

$$\frac{q \cdot m = p \cdot n}{p/q :: P \mathbin{/\!/} n \succ m} \qquad \frac{(\nexists m.\ q \cdot m = p \cdot n) \qquad P \mathbin{/\!/} n \succ m'}{p/q :: P \mathbin{/\!/} n \succ m'}$$

$$\frac{(\nexists m.\ P \mathbin{/\!/} n \succ m)}{P \mathbin{/\!/} n \triangleright n} \qquad \frac{P \mathbin{/\!/} n \succ m \qquad P \mathbin{/\!/} m \triangleright m'}{P \mathbin{/\!/} n \triangleright m'}$$

Def. FRACTRAN-halting problem

The FRACTRAN-**halting problem** is then defined as

$$\mathsf{Halt}_{\mathsf{FRACTRAN}}(P : \mathbb{L}(\mathbb{N} \times \mathbb{N}), n : \mathbb{N}) := \exists m.\ P \mathbin{/\!/} n \triangleright m.$$

Def. $\mathsf{Halt}_{\mathsf{FRACTRAN}}$

Def. regular

A FRACTRAN program P is **regular** if $\forall p/q \in P.\ q \neq 0$. For regular programs, evaluation is deterministic.

Def. FRACTRAN-computable

To define FRACTRAN computability, we fix two distinct, non-repeating sequences p_i and q_i of prime numbers. A relation $R : \mathbb{N}^k \to \mathbb{N} \to \mathbb{P}$ is FRACTRAN-**computable** if

$$\exists P : \mathbb{L}(\mathbb{N} \times \mathbb{N}).\ P \text{ is regular} \wedge$$

$$\forall n_1 \ldots n_k.\ \forall m.\ R\ (n_1, \ldots, n_k)\ m \leftrightarrow \exists j.\ P \mathbin{/\!/} q_1 \cdot p_1^{n_1} \ldots p_k^{n_k} \triangleright p_0^m \cdot j \wedge \neg \exists d.\ j = d \cdot p_0.$$

Fact 15.1. Every FRACTRAN-computable relation is functional.

15.2 Simulating counter machines in FRACTRAN

How to simulate counter machines in FRACTRAN is already hinted at by Conway [35], stating "it is usually even easier to write a FRACTRAN program to simulate other machines [than Turing machines]". The idea is straightforward. Given a program $(1, P)$, we define the encoding of states as natural numbers as follows, reusing the sequences p_i and q_i of prime numbers from before:

$$\varepsilon(c, [x_1, \ldots, x_n]) := q_c^1 \cdot p_1^{x_1} \cdots p_n^{x_n}.$$

That means we use the exponents of prime numbers as registers and have distinct registers used as Booleans indicating the current state. Note that in principle, it would also have been possible to encode the program counter as q_0^c, but verifying this translation is harder.

The compilation of a program P is defined as

$$\begin{aligned} Q &:= \gamma_1 P & \gamma_i(\mathsf{INC}\ j :: P) &:= (q_{i+1} \cdot p_j / q_i) :: \gamma_{\mathsf{S}\,i} P \\ \gamma_i[\,] &:= [\,] & \gamma_i(\mathsf{DEC}\ j\ c :: P) &:= (q_{i+1}/q_i \cdot p_j) :: (q_c/q_i) :: \gamma_{\mathsf{S}\,i} P \end{aligned}$$

It is crucial to have (q_c/q_i) appear after $(q_{i+1}/q_i \cdot p_j)$ in the compilation of DEC, because only if there are no factors p_j in the state to decrease counter j we want to jump to c. Note that if P contains an instruction $c : \mathsf{DEC}\ j\ c$, i.e. an instruction potentially looping to itself the corresponding fraction is a natural number and thus can always be chosen. We thus have to assume that P does not have such self-loops.

Fact 15.2. For every CM program P there exists a program P' with one more counter and no self-loops such that:

$$(\exists c.\ (1, P) /\!\!/ (1, v) \triangleright (c, v')) \leftrightarrow (\exists c. (1, P') /\!\!/ (1, 0 :: v) \triangleright (c, 0 :: v'))$$

Fact 15.3. The following hold:

1. q_c divides $\varepsilon(c', v)$ if and only if $c = c'$.
2. p_i divides $\varepsilon(c', v)$ if and only if $v[i] > 0$.

Theorem 15.4. The following hold:

1. If $(i, P) /\!\!/_{\mathsf{CM}} (1, v) \triangleright (c, v')$, then $\exists c'.\ Q /\!\!/_{\mathsf{FRACTRAN}} \varepsilon(1, v) \triangleright \varepsilon(c', 0 :: v')$.
2. If $Q /\!\!/_{\mathsf{FRACTRAN}} \varepsilon(1, v) \triangleright m$, then $\exists c v'. (i, P) /\!\!/_{\mathsf{CM}} (1, v) \triangleright (c, v')$.

Proof. Both parts are straightforward inductions, with some lemmas on prime numbers and division. ∎

Corollary 15.5. $\mathsf{Halt}_{\mathsf{CM}} \preceq_m \mathsf{Halt}_{\mathsf{FRACTRAN}}$

Corollary 15.6. Every CM-computable relation $R : \mathbb{N}^k \to \mathbb{N} \to \mathbb{P}$ is FRACTRAN-computable.

Proof. Immediate, since no pre- and post-processing of input and output is necessary. ∎

CHAPTER 16

Diophantine equations

Diophantine equations, named after the Greek mathematician Diophantus, are polynomial equations with natural coefficients. The question whether the problem of determining whether a Diophantine equation has a natural number solution became known as Hilbert's tenth problem. A line of work originated by Martin Davis and successfully finished by Yuri Matiyasevich culminated in the Davis-Putnam-Robinson-Matiyasevich theorem, or short DPRM-theorem, stating that a set is recursively enumerable if and only if it is Diophantine, i.e. the solution set of a polynomial.

We define Hilbert's Tenth Problem H10 and Diophantine relations in Section 16.1, show that every FRACTRAN-computable relation $R : \mathbb{N}^k \to \mathbb{N} \to \mathbb{P}$ seen as a relation $\mathbb{N}^{k+1} \to \mathbb{P}$ is Diophantine, and thereby deduce undecidability of H10 (Section 16.2). In Section 16.3 we introduce a constraint version of H10, which is easier to use as seed for undecidability proofs. In Section 16.4 we define polynomials and Hilbert's tenth problem on integers and show it undecidable.

Publications Sections 16.1 and 16.2 are from [152], Section 16.4 from [153].

[152] Larchey-Wendling and Forster. "Hilbert's Tenth Problem in Coq." *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*. 2019.

[153] Larchey-Wendling and Forster. "Hilbert's Tenth Problem in Coq (extended version)." *arXiv preprint arXiv:2003.04604*. 2019.

16.1 Definition

We define the type of polynomials over $\mathbb{N}$ with n variables and m parameters $\mathsf{poly}_{\mathbb{N}}(n, m)$ as

$$P_1, P_2 : \mathsf{poly}_{\mathbb{N}}(n, m) ::= c : \mathbb{N} \mid v : \mathbb{F}_n \mid p : \mathbb{F}_m \mid P_1 \dot{+} P_2 \mid P_1 \dot{\times} P_2.$$

Given $\nu : \mathbb{N}^n$ and $\rho : \mathbb{N}^m$ we define evaluation of polynomials as follows:

$$[\![c]\!]_{\nu,\rho} := c \qquad [\![v]\!]_{\nu,\rho} := \nu[v] \qquad [\![p]\!]_{\nu,\rho} := \rho[p]$$

$$[\![P_1 \dot{+} P_2]\!]_{\nu,\rho} := [\![P_1]\!]_{\nu,\rho} + [\![P_2]\!]_{\nu,\rho} \qquad [\![P_1 \dot{\times} P_2]\!]_{\nu,\rho} := [\![P_1]\!]_{\nu,\rho} \cdot [\![P_2]\!]_{\nu,\rho}$$

Hilbert's tenth problem is defined as

Def. Hilbert's tenth problem

$$\mathsf{H10}(n : \mathbb{N}, P_1 : \mathsf{poly}_{\mathbb{N}}(n, 0), P_2 : \mathsf{poly}_{\mathbb{N}}(n, 0)) := \exists \nu. [\![P_1]\!]_{\nu,\lambda x.0} = [\![P_2]\!]_{\nu,\lambda x.0}.$$

Def. H10

A relation $R : \mathbb{N}^k \to \mathbb{P}$ (without distinguishing input and output) is **Diophantine** if

Def. Diophantine relation

$$\exists k'. \exists P_1 P_2 : \mathsf{poly}_{\mathbb{N}}(k', k). \forall n_1 \dots n_k.$$

$$R\,(n_1, \dots, n_k) \leftrightarrow \exists \nu : \mathbb{N}^{k'}. [\![P_1]\!]_{\nu,\rho} = [\![P_2]\!]_{\nu,\rho} \text{ where } \rho := (n_1, \dots n_k).$$

A relation $R : \mathbb{N}^k \to \mathbb{N} \to \mathbb{P}$ (distinguishing input and output) is **Diophantine** if$\lambda(m, n_1, \dots, n_k).\, R(n_1, \dots, n_k)m$ is Diophantine.

16.2 FRACTRAN computation is elementary Diophantine

The results in the present section were mostly conceived and verified by Dominique Larchey-Wendling. A high-level overview is given here to keep the explanation of the translation chain self-contained. The proof uses two alternative definitions of Diophantineness: One based on elementary Diophantine constraints and one based on Diophantine logic, which is the first-order fragment of $\exists \wedge \vee$ over elementary Diophantine constraints. In this section, we omit Diophantine logic and explain the structure of the reduction based on elementary Diophantine constraints.

Elementary diophantine constraints EDIOC with a satisfaction relation $\vDash$ w.r.t. valuations $\nu, \rho : \mathbb{N} \to \mathbb{N}$ are defined as follows

$$\mathsf{econstr} ::= x \doteq c \mid x \doteq p \mid x \dot{+} y \doteq z \mid x \dot{\times} y \doteq z$$

where $x, y, z : \mathbb{N}$ denote variables and $c : \mathbb{N}$ denotes constants and $p : \mathbb{N}$ denotes parameters and:

$$\nu, \rho \vDash x \doteq c := \nu x = c \qquad \nu, \rho \vDash x \doteq p := \nu x = \rho p$$

$$\nu, \rho \vDash x \dot{+} y \doteq z := \nu x + \nu y = \nu z \qquad \nu, \rho \vDash \nu x \dot{\times} \nu y \doteq \nu z$$

We define the decision problem for elementary Diophantine constraints as

$$\mathsf{EDIOC}(C : \mathbb{L}(\mathsf{econstr}), \rho : \mathbb{N} \to \mathbb{N}) := \exists \nu : \mathbb{N} \to \mathbb{N}. \forall c \in C.\ \nu, \rho \vDash c$$

Def. elementary Diophantine relation

A relation $R : \mathbb{N}^k \to \mathbb{P}$ is **elementary Diophantine** if

$$\exists k'. \exists C : \mathbb{L}(\mathsf{econstr}). \forall n_1 \dots n_k.\ R\ (n_1, \dots, n_k) \leftrightarrow \exists \nu : \mathbb{N}^{k'}.\ \nu, (\lambda i. \textbf{if } 1 \le i \le k \textbf{ then } n_i \textbf{ else } 0) \vDash C$$

And analogously to before a relation $R : \mathbb{N}^k \to \mathbb{N} \to \mathbb{P}$ (distinguishing input and output) is **elementary Diophantine** if $\lambda(m, n_1, \dots, n_k).\ R(n_1, \dots, n_k)m$ is elementary Diophantine.

Lemma 16.1. Let $R_1 : \mathbb{N}^k \to \mathbb{N} \to \mathbb{P}$ and $R_2 : \mathbb{N}^k \to \mathbb{N} \to \mathbb{P}$ be elementary Diophantine.

The following relations are then all elementary Diophantine:

1. $\lambda(n_1, \dots, n_k).\ \top$
2. $\lambda(n_1, \dots, n_k).\ \bot$
3. $\lambda(n_1, \dots, n_k). \exists m_1 m_2.\ R_1(n_1, \dots, n_k)m_1 \wedge R_2(n_1, \dots, n_k)m_2 \wedge m_1 \le m_2$
4. $\lambda(n_1, \dots, n_k). \exists m_1 m_2.\ R_1(n_1, \dots, n_k)m_1 \wedge R_2(n_1, \dots, n_k)m_2 \wedge m_1 < m_2$
5. $\lambda(n_1, \dots, n_k). \exists m_1 m_2.\ R_1(n_1, \dots, n_k)m_1 \wedge R_2(n_1, \dots, n_k)m_2 \wedge m_1 \ne m_2$
6. $\lambda(n_1, \dots, n_k). \exists m_1 m_2.\ R_1(n_1, \dots, n_k)m_1 \wedge R_2(n_1, \dots, n_k)m_2 \wedge \nexists m'.\ m_2 = m_1 \cdot m'$

Similarly to how we defined FRACTRAN evaluation $P \mathbin{/\!/} n \triangleright m$ one can also define a small-step relation $P \mathbin{/\!/} n \succ m$ such that $P \mathbin{/\!/} n \triangleright m \leftrightarrow P \mathbin{/\!/} n \succ^* m \wedge \neg \exists m'. P \mathbin{/\!/} m \succ m'$ where $\succ^*$ denotes reflexive, transitive closure of $\succ$. One then obtains:

Corollary 16.2. The relations $\lambda(p_1, q_1, \dots, p_k, q_k, n, m.\ [(p_1, q_1), \dots, (p_k, q_k)] \mathbin{/\!/} n \succ m$ and $\lambda(p_1, q_1, \dots, p_k, q_k, n.\ \neg \exists m.\ [(p_1, q_1), \dots, (p_k, q_k)] \mathbin{/\!/} n \succ m$ are elementary Diophantine.

To extend the result to evaluation, one needs that exponentiation is elementary Diophantine, and that bounded universal quantification is exponentially elementary Diophantine. The first result was an open question for multiple years and solved by Matiyasevich [168].

[168] Matijasevič. 1970. Enumerable sets are Diophantine.

Lemma 16.3. Let $R_1, R_2 : \mathbb{N}^k \to \mathbb{N} \to \mathbb{P}$, $R_3 : \mathbb{N}^{Sk} \to \mathbb{P}$, $R : \mathbb{N} \to \mathbb{N} \to \mathbb{P}$ be elementary Diophantine. The following relations are then all elementary Diophantine:

1. $\lambda(n_1, \dots, n_k)\, m. \exists m_1 m_2. R_1(n_1, \dots, n_k) m_1 \wedge R_2(n_1, \dots, n_k) m_2 \wedge m = m_1^{m_2}$
2. $\lambda(n_1, \dots, n_k)\, m. \exists m_1 m_2. R_1(n_1, \dots, n_k) m_1 \wedge \forall m \leq m_1.\, R_3(m, n_1, \dots, n_k)$
3. $\lambda(n_1, \dots, n_k)\, m. \exists m_1 m_2. R_1(n_1, \dots, n_k) m_1 \wedge R_2(n_1, \dots, n_k) m_2 \wedge R^* m_1 m_2$

Corollary 16.4. The relations $\lambda(p_1, q_1, \dots, p_k, q_k, n, m.\ [(p_1, q_1), \dots, (p_k, q_k)] /\!/ n \succ^* m$ and $\lambda(p_1, q_1, \dots, p_k, q_k, n.\ \neg\exists m.\ [(p_1, q_1), \dots, (p_k, q_k)] /\!/ n \triangleright m$ are elementary Diophantine.

Corollary 16.5. FRACTRAN-computable relations $R : \mathbb{N}^k \to \mathbb{N} \to \mathbb{P}$ are elementary Diophantine.

Lastly, one can reduce a list of elementary constraints to a single polynomial as follows:

Theorem 16.6. Let $[(p_1, q_1), \dots, (p_n, q_n)]$ be given. Then

$$p_1 = q_1 \wedge \dots \wedge p_n = q_n \leftrightarrow 2(p_1 q_1 + \dots + p_n q_n) = p_1^2 + q_1^2 + \dots + p_n^2 + q_n^2.$$

Corollary 16.7. EDIOC $\preceq_m$ H10

Corollary 16.8. FRACTRAN-computable relations $R : \mathbb{N}^k \to \mathbb{N} \to \mathbb{P}$ are Diophantine.

Proof. Immediate, since no pre- and post-processing of input and output is necessary. ∎

16.3 Diophantine Constraints

We introduce Diophantine constraints as a slightly simpler representation of Hilbert's tenth problem. Diophantine constraints have no parameters and the only constant appearing is 1.[1]

We define constraints and constraint satisfaction for a valuation $\nu : \mathbb{N} \to \mathbb{N}$ as

$$\mathsf{constr} ::= x \doteq 1 \mid x \dot{+} y \doteq z \mid x \dot{\times} y \doteq z$$

$$\nu \vDash x \doteq 1 := \nu x = 1 \qquad \nu \vDash x \dot{+} y \doteq z := \nu x + \nu y = \nu z \qquad \nu \vDash \nu x \dot{\times} \nu y \doteq \nu z := \nu x \cdot \nu y = \nu z$$

Def. Diophantine constraint solvability

Diophantine constraint solvability is defined as

$$\mathsf{H10C}(C : \mathbb{L}\,\mathsf{constr}) := \exists \nu : \mathbb{N} \to \mathbb{N}. \forall c \in C.\ \nu \vDash c$$

We reduce elementary constraint solvability to H10C by encoding constants c as variables x_c via the equations $x_0 \doteq 0, x_1 \doteq 1, x_2 \doteq 1 + x_1, \dots, x_c \doteq 1 + x_{c-1}$.

Fact 16.9. For every n one can compute C_n such that $\nu \vDash C_n \leftrightarrow \forall i < n. \nu i = i$.

One can compute the maximal constant appearing in an elementary constraint list after applying a parameter valuation ρ:

Fact 16.10. There is a function maxc_ρ s.t. $\forall p \doteq x \in C.\ \rho p < \mathsf{maxc}_\rho\ C$ and $\forall c \doteq x \in C.\ c < \mathsf{maxc}_\rho\ C$.

Theorem 16.11. EDIOC $\preceq_m$ H10C

[1] The naming scheme is a bit unfortunate: Diophantine constraints are *more elementary* than elementary Diophantine constraints. We however follow the scheme already published in the literature.

Proof. Let the constraint translation function γ be defined as

$$\gamma_{\rho,m}(y \doteq x) := (m+y)\dot{+}0 \doteq (m+x) \qquad \gamma_{\rho,m}(x_1 \dot{+} x_2 \doteq x) := (m+x_1)\dot{+}(m+x_2) \doteq (m+x)$$

$$\gamma_{\rho,m}(p \doteq x) := \rho p \dot{+} 0 \doteq (m+x) \qquad \gamma_{\rho,m}(x_1 \dot{\times} x_2 \doteq x) := (m+x_1)\dot{\times}(m+x_2) \doteq (m+x)$$

The reduction function is $f(C,\rho) := C_{1+\mathsf{maxc}_\rho C} +\!\!+ \mathsf{map}\ \gamma_{\rho,1+\mathsf{maxc}_\rho C}\ C$. Let $m := 1 + \mathsf{maxc}_\rho C$.

For the direction from left to right, let $\nu, \rho \vDash C$. Then $\lambda i.\mathbf{if}\ i < m\ \mathbf{then}\ \nu i\ \mathbf{else}\ \nu(i-m) \vDash f(C,\rho)$. For the other direction, let $\nu \vDash f(C,\rho)$. Then $(\lambda i.\nu(m+i)), \rho \vDash C$. ∎

16.4 Hilbert's tenth problem over integers

We prove Hilbert's tenth problem over integers $\mathsf{H10}_\mathbb{Z}$ undecidable by reduction from H10. We define the type of polynomials over integers with n variables and m parameters $\mathsf{poly}_\mathbb{N}(n,m)$ as

$$P_1, P_2 : \mathsf{poly}_\mathbb{Z}(n,m) ::= c : \mathbb{Z} \mid \nu : \mathbb{F}_n \mid p : \mathbb{F}_m \mid P_1 \dot{+} P_2 \mid P_1 \dot{\times} P_2$$

Given $\nu : \mathbb{Z}^n$ and $\rho : \mathbb{Z}^m$ we define evaluation of polynomials as

$$[\![c]\!]_{\nu,\rho} := c \qquad [\![\nu]\!]_{\nu,\rho} := \nu[\nu] \qquad [\![p]\!]_{\nu,\rho} := \rho[p]$$

$$[\![P_1 \dot{+} P_2]\!]_{\nu,\rho} := [\![P_1]\!]_{\nu,\rho} + [\![P_2]\!]_{\nu,\rho} \qquad [\![P_1 \dot{\times} P_2]\!]_{\nu,\rho} := [\![P_1]\!]_{\nu,\rho} \cdot [\![P_2]\!]_{\nu,\rho}$$

Def. Hilbert's tenth problem over integers

Hilbert's tenth problem over integers is defined as

Def. $\mathsf{H10}_\mathbb{Z}$

$$\mathsf{H10}_\mathbb{Z}(n : \mathbb{N}, P : \mathsf{poly}_\mathbb{Z}(n,0)) := \exists \nu.\ [\![P]\!]_{\nu,\lambda x.0} = 0$$

The core of the reduction is to show that the natural numbers are a Diophantine subset of the integers, i.e. to characterise non-negativity of integers as Diophantine equation over integers, which is a direct consequence of Lagrange's theorem:

Fact 16.12. $\forall z{:}\mathbb{Z}.\ z \geq 0 \leftrightarrow \exists abcd : \mathbb{N}.\ z = a^2 + b^2 + c^2 + d^2$

Theorem 16.13. $\mathsf{H10} \preceq_m \mathsf{H10}_\mathbb{Z}$

Proof. Let $f : \mathsf{poly}_\mathbb{Z}(n,0) \to \mathsf{poly}_\mathbb{Z}(4\cdot n, 0)$ be the function replacing every variable $i : \mathbb{F}_n$ in a polynomial by the polynomial expression

$$((4\cdot n)\dot{\times}(4\cdot n))\dot{+}((4\cdot n+1)\dot{\times}(4\cdot n+1))\dot{+}((4\cdot n+2)\dot{\times}(4\cdot n+2))\dot{+}((4\cdot n+3)\dot{\times}(4\cdot n+3)).$$

Now given polynomials $P_1, P_2 : \mathsf{poly}_\mathbb{N}(n,0)$, the reduction function returns the polynomial $f P_1 \dot{+} (-1) \dot{\times} f(P_2) : \mathsf{poly}_\mathbb{Z}(4\cdot n, 0)$. If $\mathsf{H10}(n, P_1, P_2)$, $\mathsf{H10}_\mathbb{Z}(f P_1 \dot{+} (-1) \dot{\times} f(P_2))$ follows by Fact 16.12. If $[\![f P_1 \dot{+} (-1) \dot{\times} f(P_2)]\!]_{\nu,\lambda x.0} = 0$ with $\nu = [a_1, b_1, c_1, d_1, \ldots, a_n, b_n, c_n, d_n]$, then in particular $[\![f P_1]\!]_{\nu,\lambda x.0} = [\![f P_2]\!]_{\nu,\lambda x.0}$ and thus $[\![P_1]\!]_{\nu',\lambda x.0} = [\![P_2]\!]_{\nu',\lambda x.0}$ with $\nu' := [a_1^2 + b_1^2 + c_1^2 + d_1^2, \ldots, a_n^2 + b_n^2 + c_n^2 + d_n^2]$. ∎

CHAPTER 17

μ-recursive functions

The notion of μ-recursive functions, also known as general or partial recursive functions, was formally introduced by Gödel [93], who conceived the definition based on discussions with Herbrand [107, 108]. Programming in μ-recursive functions is combinatorial: There are no variables, but functions like successor or projection are composed via a composition operator. Recursion is enabled by an unbounded minimisation operator μf, returning the least zero of f. The input of μ-recursive functions have explicit input and output, respectively of type $\mathbb{N}^k$ and $\mathbb{N}$.

[93] Gödel. 1934. On formally undecidable propositions of Principia Mathematica and related systems.

[107] Herbrand. 1930. Les bases de la logique Hilbertienne.

[108] Herbrand. 1932. Sur la non-contradiction de l'Arithmétique.

In Section 17.1 we define μ-recursive functions and their halting problem. In Section 17.2 we show that Diophantine relations are μ-recursive.

Publications This chapter is based on

[153] Larchey-Wendling and Forster. "Hilbert's Tenth Problem in Coq (extended version)." *arXiv preprint arXiv:2003.04604*. 2019.

17.1 Definition

We use the syntax of μ-recursive functions computing a partial function of type $\mathbb{N}^k \rightharpoonup \mathbb{N}$ defined as dependent inductive type func_k by Larchey-Wendling [150]:

[150] Larchey-Wendling. 2017. Typing total recursive functions in Coq.

$$\frac{c : \mathbb{N}}{\mathsf{cst}\, c : \mathsf{func}_0} \qquad \frac{}{\mathsf{zero} : \mathsf{func}_1} \qquad \frac{}{\mathsf{succ} : \mathsf{func}_1} \qquad \frac{j : \mathbb{F}_k}{\mathsf{proj}\, j : \mathsf{func}_k} \qquad \frac{f : \mathsf{func}_i \qquad G : (\mathsf{func}_k)^i}{\mathsf{comp}\, f\, G : \mathsf{func}_k}$$

$$\frac{f : \mathsf{func}_k \qquad g : \mathsf{func}_{2+k}}{\mathsf{primrec}\, f\, g : \mathsf{func}_{1+k}} \qquad \frac{f : \mathsf{func}_{1+k}}{\mu f : \mathsf{func}_k}$$

The constructor $\mathsf{cst}\, c$ is used for constants, zero is the constant 0 function, succ is the successor function, $\mathsf{proj}\, j$ is the projection on the j-th component of argument $\mathbb{N}^k$, comp is function composition, $\mathsf{primrec}$ is primitive recursion, and μ is unbounded minimisation. Note that the constructor comp makes func_k a nested inductive type, due to the use of vectors of type $(\mathsf{func}_k)^i$. The evaluation relation $\triangleright : \mathsf{func}_k \to \mathbb{N}^k \to \mathbb{N} \to \mathbb{P}$ is defined in Figure 17.1.

We define the **halting problem for μ-recursive functions** as

Def. halting problem for μ-recursive functions

$$\mathsf{Halt}_\mu(k : \mathbb{N}, f : \mathsf{func}_k, v : \mathbb{N}^k) := \exists x. f(v) \triangleright x.$$

Def. Halt_μ

A relation $R : \mathbb{N}^k \to \mathbb{N} \to \mathbb{P}$ is **μ-recursive** if

Def. μ-recursive

$$\exists f : \mathsf{func}_k. \forall n_1 \dots n_k. \forall m. R\,[n_1, \dots, n_k]\, m \leftrightarrow f\,[n_1, \dots, n_k] \triangleright m.$$

Note that for μ-recursive functions we could – in contrast to Turing machines or the λ-calculus – define μ-recursive computability of a relation directly as inductive predicate, omit-

$$\frac{}{\mathsf{cst}\, n\,(v) \triangleright n} \qquad \frac{}{\mathsf{zero}(v) \triangleright 0} \qquad \frac{}{\mathsf{succ}([x]) \triangleright 1 + x} \qquad \frac{}{\mathsf{proj}\, j\,(v) \triangleright v[j]}$$

$$\frac{(\forall j : \mathbb{F}_k.\ G[j](v) \triangleright w[j]) \qquad f(w) \triangleright x}{\mathsf{comp}\, f\ G\,(v) \triangleright x} \qquad \frac{f(x :: v) \triangleright 0 \qquad \forall j : \mathbb{F}_x. f(j :: v) \triangleright 1 + w[j]}{\mu f \triangleright x}$$

$$\frac{f(v) \triangleright x}{\mathsf{primrec}\, f\ g\,(0 :: v) \triangleright x} \qquad \frac{\mathsf{primrec}\, f\ g\,(n :: v) \triangleright y \qquad g(n :: y :: v) \triangleright x}{\mathsf{primrec}\, f\ g\,(1 + n :: v) \triangleright x}$$

Figure 17.1.: Evaluation for μ-recursive functions

[26] Carneiro. 2019. Formalizing Computability Theory via Partial Recursive Functions.

ting the need for syntax, similar to Carneiro [26, Figure 5]. We however follow the textbook approach here, to be more in line with the other presentations.

17.2 Diophantine relations are μ-recursive

The construction essentially consists of two auxiliary and total μ-recursive functions: A function e evaluating polynomials, a function t testing whether two natural numbers are equal, and a functions p_i translating a natural number to a vector $v : \mathbb{N}^n$ and projecting out the i-th component.

Fact 17.1. Given $P : \mathsf{poly}(n, m)$ there is $e : \mathsf{func}_{n+m}$ such that $e(v +\!\!+ \rho) \triangleright [\![P]\!]_{v,\rho}$.

Fact 17.2. There is $t : \mathsf{func}_2$ such that $\forall x_1, x_2.\exists x.\ t([x_1, x_2]) \triangleright x \wedge (x_1 = x_2 \leftrightarrow x = 0)$.

Fact 17.3. Given n there are $p_0, \ldots, p_{n-1} : \mathsf{func}_1$ such that $\forall v : \mathbb{N}^n.\exists x.; \forall i.\ p_i(v) \triangleright v[i]$.

Theorem 17.4. Every functional Diophantine relation $R : \mathbb{N}^k \to \mathbb{N} \to \mathbb{P}$ is μ-recursive.

Proof. Let R, k'. and $P_1, P_2 : \mathsf{poly}(\mathsf{S}\, k, k')$ be given s.t. for all $n_1, \ldots, n_k$ and m:

$$R\ (n_1, \ldots, n_k)\ m \leftrightarrow \exists \rho : \mathbb{N}^{k'}.\ [\![P_1]\!]_{v,\rho} = [\![P_2]\!]_{v,\rho} \ \text{ where } v := [m, n_1, \ldots n_k]$$

We first derive polynomials $P_1', P_2' : \mathsf{poly}(k, \mathsf{S}\, k')$ remapping the first parameter in P_1 and P_2 to a variable, i.e. $\forall m\, v \rho.\ [\![P_i]\!]_{m::v,\rho} = [\![P_i']\!]_{v,m::\rho}$.

Now e_1, e_2 evaluate P_1', P_2' as in Fact 17.1. We define $f : \mathsf{func}_k$ as follows:

$$\mathsf{comp}\, p_0\, (\mu\, (\mathsf{comp}\, t\, \big[\mathsf{comp}\, e_1 [\mathsf{proj}\ \ 1, \ldots, \mathsf{proj}\ \ k, \mathsf{comp}\, p_1\, (\mathsf{proj}\, 0), \ldots, \mathsf{comp}\, p_{k'}\, (\mathsf{proj}\, 0)],$$
$$\mathsf{comp}\, e_2 [\mathsf{proj}\ \ 1, \ldots, \mathsf{proj}\ \ k, \mathsf{comp}\, p_1\, (\mathsf{proj}\, 0), \ldots, \mathsf{comp}\, p_{k'}\, (\mathsf{proj}\, 0)]\big]))$$

The test function passed to μ is a $1 + k$ ary function. The k arguments are the k inputs of f, whereas the additional argument x is decomposed into $1 + k'$ arguments for the search. If the test succeeds on x, the minimisation μ returns x, and we return the first projection of x. ∎

Corollary 17.5. $\mathsf{H10} \preceq_m \mathsf{Halt}_\mu$

Part III
Undecidability reductions

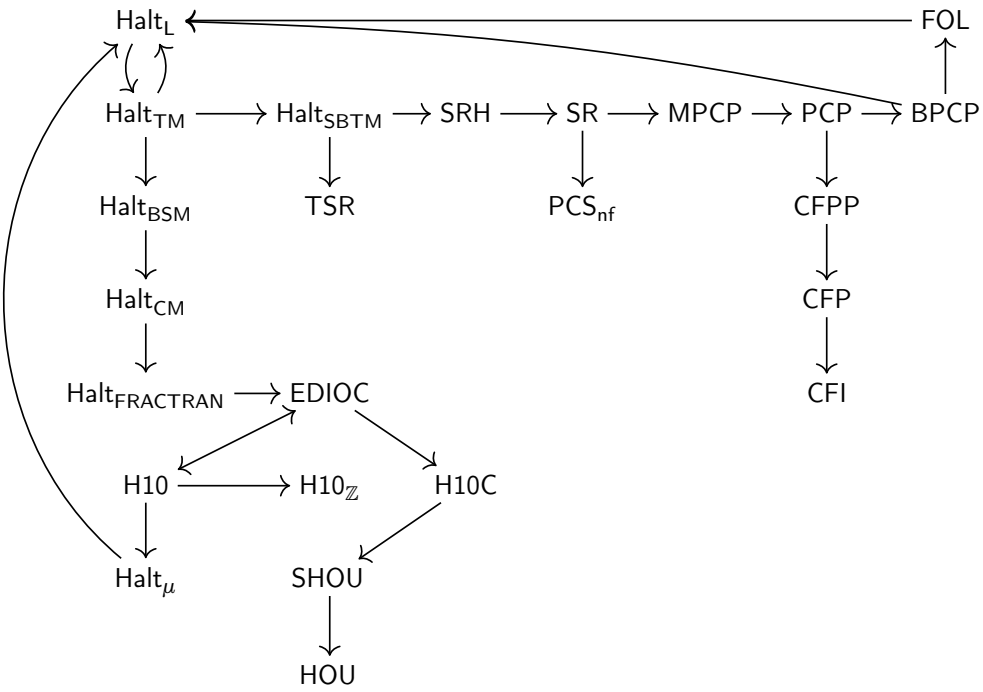


Figure 18.1.: Problems covered in Parts II and III.

CHAPTER 18

Introduction: Undecidability reductions

Machine-checked decidability proofs can be frequently found in the literature [58, 163, 203, 57, 33]. Usually, such proofs work in constructive foundations such as CIC underlying the Coq proof assistant and define decidability in terms of boolean functions of type $X \to \mathbb{B}$. We used such synthetic definitions in Part I as well, relying on the fact that all functions definable in CIC correspond to a program in a Turing-complete model of computation. For undecidability proofs, a similarly simple approach fails: Since CIC is consistent with axioms entailing the decidability of every problem, an axiom-free proof of the absence of a decision function is impossible.

[58] Doczkal and Smolka. 2014. Completeness and Decidability Results for CTL in Coq.

[163] Maksimović and Schmitt. 2015. HOCore in Coq.

[203] Schäfer, Smolka, and Tebbi. 2015. Completeness and decidability of de Bruijn substitution algebra in Coq.

[57] Doczkal and Bard. 2018. Completeness and decidability of converse PDL in the constructive type theory of Coq.

[33] Cohen and Mahboubi. 2010. A Formal Quantifier Elimination for Algebraically Closed Fields.

However, most undecidability proofs of machine-independent problems in the literature are by reduction rather than directly exploiting a logical paradox. To prove the undecidability of an involved problem, reduction chains are used which stepwise reduce a problem already known to be undecidable to the problem in question. We make use of this technique to establish the undecidability of problems in CIC without assuming axioms. Concretely, we establish a notion $\mathcal{U}p$ for predicates $p : X \to \mathbb{P}$ such that for any p undecidable by (constructive) reduction from the halting problem, $\mathcal{U}p$ can be proved without the assumption of axioms. In Part I we have discussed the axiom CT, stating that every function $f : \mathbb{N} \to \mathbb{N}$ is computable in a Turing-complete model of computation from Part II. We will see that assuming CT implies $\mathcal{U}p \leftrightarrow \neg\mathcal{D}p$.

$\mathcal{D}p$ → Sec. 4.1, Page 31

To demonstrate the power of the approach, we cover the problems which build the historical basis of undecidability, also displayed in Figure 18.1: (Semi-Thue) string-rewriting, the word problem for semi-groups, Post's correspondence problem, the intersection and palindrome problem for context-free grammars, the Entscheidungsproblem, and higher-order unification in the simply-typed Curry-style λ-calculus. The reduction corollaries deduced in Part II from the simulation results all can also be seen as synthetic undecidability proofs for the covered halting problems and H10.

The contributed undecidability proofs are the first machine-checked undecidability proofs for machine-independent problems. Textbooks usually only outline proof ideas, but do not give fully verified proofs and in particular omit constructions of programs in models of computations. As in Part II, a crucial part of the contribution is thus again to identify the inductive invariants sufficient to make proofs by induction possible. The constructions of programs in models are omitted in our approach as well, but in contrast to textbooks this approach is sound-by-construction and omissions do not have to be individually checked to trust a proof. Again a significant proof engineering effort is involved, in total comprising 6000 lines of code (for more details see Section 18.3).

The problems covered in this part also form the basis for the Coq Library of Undecidability Proofs, on which we report in Chapter 25. This part is written in encyclopedic style and can be read as introduction to the basic problems of the library, with self-contained explanation of their definitions.

18.1 Outline with historical references

In 1914, Axel Thue defined **Semi-Thue and Thue string rewriting** SR and TSR [225]. Semi-Thue string rewriting is first-order string rewriting, i.e. the reflexive transitive closure of a directed rewriting relation $x \succ_R y$. Thue string rewriting is the reflexive, symmetric, transitive closure of this relation. In 1947, Emil Post [191] and Andrey Markov [165] independently proved the undecidability of TSR. Post's proof is by reduction from Turing machines, whereas Markov's uses a reduction from Post canonical systems. We prove (Semi-Thue) string rewriting SR undecidable by reduction from the halting problem of simple binary Turing machines $\mathsf{Halt}_{\mathsf{SBTM}}$.

[225] Thue. 1914. Probleme uber Veränderungen von Zeichenreihen nach gegebenen Regeln.

[191] Post. 1947. Recursive unsolvability of a problem of Thue.

[165] Markov. 1951. Impossibility of Certain Algorithms in the Theory of Associative Systems.

In the 1920s, Post introduced **canonical systems**, which are a more general form of string rewriting. In 1943, Post [188] proved that canonical systems can be brought into a normal form, and that the corresponding rewriting problem $\mathsf{PCS}_{\mathsf{nf}}$ is undecidable. We prove $\mathsf{PCS}_{\mathsf{nf}}$ undecidable by reduction from SR.

[188] Post. 1943. Formal reductions of the general combinatorial decision problem.

In 1945, Post [190] defined the **Post correspondence problem** PCP and proved it undecidable by reduction from $\mathsf{PCS}_{\mathsf{nf}}$. We define a boolean and a modified variant of PCP (BPCP and MPCP) [113], and prove many-one reductions $\mathsf{SR} \preceq_m \mathsf{MPCP} \preceq_m \mathsf{PCP} \preceq_m \mathsf{BPCP}$, inspired by a reduction by Davis [48].

[190] Post. 1946. A variant of a recursively unsolvable problem.

[113] Hopcroft, Motwani, and Ullman. 2006. Introduction to Automata Theory, Languages, and Computation (3rd Edition).

[48] Davis and Weyuker. 1983. Computability, complexity, and languages - fundamentals of theoretical computer science.

In 1956, **context-free grammars** were introduced by Noam Chomsky [29]. In 1961, Bar-Hillel, Perles, and Shamir [7] proved that the intersection problem CFI of context-free grammars is undecidable by reduction from PCP. It is folklore that the reduction can also be factored by first showing that PCP can be reduced to the problem CFP of determining whether a context-free grammar contains a palindrome, and then reducing CFP to CFI. We factor the proof via an intermediate problem: the palindrome problem for Post grammars CFPP, which is a special form of linear grammars suitable for reduction from PCP, and show $\mathsf{PCP} \preceq_m \mathsf{CFPP} \preceq_m \mathsf{CFP} \preceq_m \mathsf{CFI}$.

[29] Chomsky. 1956. Three models for the description of language.

[7] Bar-Hillel et al.. 1961. On formal properties of simple phrase structure grammars.

In 1936, the undecidability of the **Entscheidungsproblem of first-order logic** was independently published by Church [31] (by reduction from the halting problem of the λ-calculus) and Turing [229] (by reduction from the printing problem of Turing machines, see footnote 4 on page 107). Already in the 1920s, the unsolvability of the Entscheidungsproblem was anticipated by Post [192], by reduction from canonical systems. We give an undecidability proof for various formulations of the Entscheidungsproblem for classical, intuitionistic, and minimal first-order logic by reduction from BPCP.

[31] Church. 1936. An unsolvable problem of elementary number theory.

[229] Turing. 1936. On computable numbers, with an application to the Entscheidungsproblem.

[192] Post. 1994. Absolutely unsolvable problems and relatively undecidable propositions–Account of an anticipation (1941).

In 1972 and 1973, Claudio Lucchesi [160] and Gérard Huet [117] independently proved the undecidability of third-order unification in the full, Curry-style, simply-typed λ-calculus. In 1983, William Goldfarb [96] improved the result to second-order unification with one binary constant. We prove the general **higher-order unification** problem for the full, Curry-style, simply typed λ calculus HOU undecidable by reduction from Hilbert's tenth problem H10, following a proof employing Church encodings by Giles Dowek [60].

[160] Lucchesi. 1972. The undecidability of the unification problem for third order languages.

[117] Huet. 1973. The undecidability of unification in third order logic.

[96] Goldfarb. 1981. The undecidability of the second-order unification problem.

[60] Dowek. 2001. Higher-Order Unification and Matching..

The problems covered in this part of the thesis, together with the halting problems covered in Part II, are depicted in Figure 18.1.

18.2 Related work

Machine-checked undecidability proofs of machine-independent problems have – to the best of our knowledge – not been considered previous to the proofs described here. However,

various machine-checked undecidability results now depend on our results. We discuss these results in Section 25.2.

We have already discussed related work regarding reductions between halting problems of models of computation in Section 10.2.

18.3 Mechanisation in Coq

The results of the present chapter are all contributed to the Coq Library of Undecidability Proofs. They are also part of the following repository:

`https://ps.uni-saarland.de/~forster/thesis`

The Coq code concerning the present chapter comprises around 6.300 lines code. About 20% of the code was contributed by Simon Spies as part of his Bachelor's thesis advised by the author of this thesis.

Once again, the key feature of Coq used is setoid rewriting [214]. In contrast to Part II, the choice of proof assistant is crucial for the meaningfulness of results. Our definition of synthetic undecidability relative to a halting problem only works in a constructive foundation, ruling out e.g. HOL-based proof assistants. If one would like to verify synthetic undecidability in Lean, the `noncomputable` keyword has to be avoided.

[214] Sozeau, Matthieu; Harvard University. 2009. A New Look at Generalized Rewriting in Type Theory.

The central theorems in this part of the pdf of this thesis are hyperlinked with the html-version of the Coq code, indicated by a clickable 🔗-symbol.

CHAPTER 19

Synthetic Undecidability

We here discuss a notion of synthetic undecidability relative to a halting problem which can be established without the use of any axioms. The key observation is that any proof of the absence of a decision function (i.e. $\neg\mathcal{D}p$) by reduction can be factored into a proof of a property $\mathcal{U}p$ such that in general $\neg\mathcal{D}p \to \mathcal{U}p$ and $\mathsf{CT} \to \mathcal{U}p \to \neg\mathcal{D}p$.

19.1 Definition

We fix a halting problem Halt from one of the models introduced in Part II, for instance $\mathsf{Halt} := \mathsf{Halt}_{\mathsf{TM}_1}$. A problem $p : X \to \mathbb{P}$ is called **synthetically undecidable relative to** Halt if its decidability implies that Halt is co-enumerable:

Def. synthetic undecidability

$$\mathcal{U}p := \mathcal{D}p \to \mathcal{E}(\overline{\mathsf{Halt}})$$

We emphasise the term *relative*, since the notion depends on the definition of Halt. While we use a halting problem from Part II, one can obtain a weaker notion by replacing Halt with a solution for Post's problem for Turing machines or another model from Part II. A solution of Post's problem via the Friedberg-Muchnik theorem [85, 172] is a predicate p such that Halt does not Turing-reduce to p, meaning decidability of p does not imply decidability of Halt. Thus, such a definition would be weaker, since a synthetic undecidability proof respective i.e. the new definition is weaker. One could also use $\mathcal{D}(\mathsf{Halt})$ or $\mathcal{D}(\overline{\mathsf{Halt}})$ in the conclusion of $\mathcal{U}p$, but we pick $\mathcal{E}(\overline{\mathsf{Halt}})$ since it is the weakest condition of the three.

We will leave out both "relatively" and "synthetically" when talking about undecidability in this part of the thesis, since no other notion of undecidability other than $\mathcal{U}p$ appears.

In general, we are only interested in problems on enumerable and discrete types X. Occasionally, we will have that X is not provably discrete and enumerable, since we have that X is a dependent type, e.g. the type of closed terms $X := \Sigma t : \mathsf{tm}_{\mathsf{L}}.\ \mathsf{closed}\ t$. Such a type X is enumerable, but to prove discreteness one would have to reformulate the definition of closedness or assume proof irrelevance PI.

proof irrelevance
→ Sec. 7.3, Page 70

Based on the results established in Chapter 4, it is straightforward to prove the following closure properties:

Fact 19.1. $\mathcal{U}(\mathsf{Halt}_{\mathsf{TM}_1})$

Proof. Since $\mathcal{D}(\mathsf{Halt}_{\mathsf{TM}_1}) \to \mathcal{E}(\overline{\mathsf{Halt}_{\mathsf{TM}_1}})$ by Facts 4.5 and 4.14. ∎

Fact 19.2. $\mathcal{U}\overline{p} \to \mathcal{U}p$

Proof. Let $\mathcal{D}\overline{p} \to \mathcal{E}(\overline{\mathsf{Halt}})$ and $\mathcal{D}p$. Then $\mathcal{D}\overline{p}$ by Corollary 4.5. Thus $\mathcal{E}(\overline{\mathsf{Halt}})$ as wanted. ∎

Lemma 19.3. If $\mathcal{D}q \to \mathcal{D}p$ then $\mathcal{U}p \to \mathcal{U}q$.

Proof. Let $\mathcal{D}q \to \mathcal{D}p$, $\mathcal{D}p \to \mathcal{E}(\overline{\mathsf{Halt}})$, and $\mathcal{D}q$. By the first and third assumption we have $\mathcal{D}p$, and by the second $\mathcal{E}(\overline{\mathsf{Halt}})$ as wanted. ∎

Note that one could weaken the definition of $\mathcal{U}p$ as follows:

$$\mathcal{U}'\,p := \neg\mathcal{E}(\overline{\mathsf{Halt}}) \to \neg\mathcal{D}p$$

We have that $\mathcal{U}p \to \mathcal{U}'\,p$, but the converse direction requires classical logic. The weaker definition $\mathcal{U}'\,p$ has the advantage that it has a negative conclusion, and thus case arbitrary distinctions are possible to start the proof. However, the stronger version $\mathcal{U}p$ suffices for all our use cases, and has a more constructive flavour, ruling out arbitrary case distinctions.

19.2 Synthetic Undecidability by Reduction

Undecidability proofs are rarely given directly by exhibiting a logical contradiction. Usually, one derives a logical contradiction from the assumption of enumerability of the complement of the self-halting problem. Afterwards, the halting problems and other problems are proved undecidable **by reduction**.

In fact, most published results on machine-independent undecidable problems use **many-one reductions** as defined in Section 5.1. Since many-one reductions transport decidability backwards, we have:

Lemma 19.4. If $p \preceq_m q$ and p is undecidable, then q is undecidable.

Proof. Immediate using Fact 5.2. ∎

Using the results from the last chapter, we immediately have:

Corollary 19.5. $\mathcal{U}(\mathsf{Halt}_{\mathsf{TM}})$, $\mathcal{U}(\mathsf{Halt}_{\mathsf{BSM}})$, $\mathcal{U}(\mathsf{Halt}_{\mathsf{CM}})$, $\mathcal{U}(\mathsf{Halt}_{\mathsf{FRACTRAN}})$, and $\mathcal{U}(\mathsf{Halt}_{\mu})$.

Corollary 19.6. $\mathcal{U}(\mathsf{H10})$, $\mathcal{U}(\mathsf{H10C})$, and $\mathcal{U}(\mathsf{H10}_{\mathbb{Z}})$.

We deduce $\mathcal{U}(\mathsf{Halt}_{\mathsf{L}})$ from a reduction $\mathsf{Halt}_{\mathsf{TM}} \preceq_m \mathsf{Halt}_{\mathsf{L}}$ and the already established reduction $\mathsf{Halt}_{\mathsf{TM}_1} \preceq_m \mathsf{Halt}_{\mathsf{TM}}$ in Section 28.1.

We can also show the undecidability of a problem formulated purely in terms of functions:

Lemma 19.7. The predicate $\mathcal{K}_{\mathbb{N}\to\mathbb{B}}(f : \mathbb{N} \to \mathbb{B}) := \exists n.\, f\,n = \mathsf{true}$ is undecidable.

Proof. We show that $\mathsf{Halt}_{\mathsf{TM}_1} \preceq_m \mathcal{K}_{\mathbb{N}\to\mathbb{B}}$, which suffices by Fact 19.1 and Lemma 19.4. The reduction function takes as input a single tape Turing machine M, a tape t and maps them to the function of type $\mathbb{N} \to \mathbb{B}$ which takes n, runs M on t for n steps, and outputs true iff the computation halted. ∎

Note that the proof inlines a semi-decidability proof for $\mathsf{Halt}_{\mathsf{TM}}$ and Fact 5.10. One can prove the undecidability of similar problems on functions. Since we focus on problems on enumerable, discrete types and $\mathbb{N} \to \mathbb{B}$ is neither, we do not develop more such results.

We are only aware of two non-artificial problems which are not shown undecidable by many-one reductions: Kolmogorov complexity [138, 139] is only truth-table reducible from the halting problem [146]. The undecidability proof for semi-unification [126] was first given by weak truth-table reduction from Turing machine immortality [112]. While there might be

[138] Kolmogorov. 1963. On tables of random numbers.

[139] Kolmogorov. 1965. Three approaches to the quantitative definition of information.

[146] Kummer. 1996. On the complexity of random strings.

[126] Kfoury, Tiuryn, and Urzyczyn. 1993. The undecidability of the semi-unification problem.

[62] Dudenhefner. 2021a. Constructive Many-one Reduction from the Halting Problem to Semi-unification.

an inherent need for weak truth-table reducibility if factoring the proof via Turing machine immortality, a recent simplified undecidability proof by Dudenhefner [62] is by many-one reduction from Turing machine halting via a uniform boundedness problem.

Since truth-table reducibility also transports decidability backwards we have:

truth-table reduciblity $\preceq_{\mathrm{tt}}$
→ Sec. 5.4, Page 50

Lemma 19.8. If $p \preceq_{\mathrm{tt}} q$ and p is undecidable, then q is undecidable.

Proof. Immediate using Lemma 5.25. ∎

We omit similar results regarding bounded Turing reducibility or equivalently weak truth-table reducibility because they require MP and immediately state the how to recover a synthetic undecidability proof by relying on Turing reductions as defined in Chapter 9.

Markov's principle MP
→ Sec. 3.1, Page 20

Turing-reduciblity
→ Sec. 9.1, Page 95

Lemma 19.9. If $p \preceq_{\mathrm{T}} q$, MP holds, and p is undecidable, then q is undecidable.

Proof. Immediate using Lemma 9.5. ∎

Note how we require the axiomatic assumption of Markov's principle MP in this theorem: without MP, Turing reductions do not transport decidability backwards.

In the setting of synthetic undecidability, assuming axioms is never harmless, since some axioms might allow the definition of non-computable functions.

19.3 Axioms

When assuming axioms like the axiom of choice and the law of excluded middle together in CIC, one can prove that every problem is synthetically decidable. Thus, a synthetic study of undecidability can only work with no or restricted axioms available. In general, every axiom which is consistent with CT as discussed in Chapter 7 is not problematic.

Recall that Markov's principle MP, functional extensionality, and propositional extensionality are consistent with CT and can thus be freely assumed in synthetic undecidability proofs. For stronger axioms like the law of excluded middle there is still good reason to believe that they can be assumed for synthetic undecidability arguments. Lacking formal proof of this belief, we outline how one can obtain sound undecidability proofs in the presence of axioms, by focusing on undecidability by reduction.

The idea is to define reducibility notions $\preceq^A$, where A is an axiom, such that to establish $p \preceq^A q$ one has to give the reduction function fully-constructively, but is allowed to use A for the verification of the reduction.

We define that a problem $p : X \to \mathbb{P}$ is many-one reducible to $q : Y \to \mathbb{P}$ under axiom A if

$$p \preceq_m^A q := \exists f : X \to Y.\, A \to \forall x.\, px \leftrightarrow q(f\,x).$$

Now if e.g. LEM implies that p is undecidable by many-one reduction from the halting problem, we can prove $\mathsf{Halt}_{\mathsf{TM}} \preceq_m^{\mathsf{LEM}} p$. Since the axiom A is only used in the proof and not for the construction of the function f, we know that f still corresponds to e.g. a Turing machine, which however now has a classical correctness proof.

One can introduce similar notions $p \preceq_{\mathrm{tt}}^A q$ and $p \preceq_{\mathrm{T}}^A q$, but we refrain from doing so here because we do not use debatable axioms, but only use functional extensionality to simplify the treatment of binders in syntax.

19.4 Other proof assistants

We assess that at the time of writing, the Coq proof assistant is the only prominent system in which undecidability proofs can be feasibly mechanised. While proof assistants based on classical HOL such as Isabelle/HOL cannot guarantee the computability of functions for free, other type theory-based proof assistants also seem not to scale. Agda does not support the various complex automation techniques necessary for undecidability proofs, especially regarding automatic case analysis and arithmetic reasoning. Lean could be used in principle as it is also built on top of CIC, but Lean's standard library decidedly uses classical logic and choice operators already for basic results on natural numbers.

CHAPTER 20

String rewriting systems

String rewriting problems are amongst the oldest decision problems. In their modern form, string rewriting systems are usually presented as finite sets of rules, where a rule consists of two strings (u, v). If $x = x_1 u x_2$ and there is a rule (u, v), then x can be rewritten to $x_1 v x_2$.

[225] Thue. 1914. Probleme uber Veränderungen von Zeichenreihen nach gegebenen Regeln.

A first systematic treatment of string rewriting problems goes back to Thue [225], who posed the decidability of the word problem for Thue systems as an open question.[1] Thue Systems are string rewriting systems where all rules can be applied both ways. Equivalently, the word problem for Thue systems (TSR) can be seen as deciding the equality of two terms in a finitely presentable semigroup, the so-called *word problem for semigroups*.

In the same paper, Thue introduced Semi-Thue systems, which are nowadays simply known as string rewriting systems. The SR decision problem asks whether a string can be reached from another string with rewrites from a given set of rules.

[191] Post. 1947. Recursive unsolvability of a problem of Thue.

[165] Markov. 1951. Impossibility of Certain Algorithms in the Theory of Associative Systems.

[188] Post. 1943. Formal reductions of the general combinatorial decision problem.

In 1947, Emil Post [191] and Andrey Markov [165] independently proved the undecidability of TSR, which implies the undecidability of SR. Post's proof reduces the printing problem of Turing machines to TSR (see footnote 4 on page 107), whereas Markov's proof is by reduction from Post canonical systems [188]. Post canonical systems were devised by Post in the 1920s and are a more general and complicated form of rewriting. In his 1943 paper, Post immediately showed that canonical systems can be brought into a considerably less complicated normal form.

In this chapter we cover the word problem for Thue systems (TSR), i.e. reachability in rewriting systems with a symmetric set of rules, the generalised halting problem for string rewriting systems (SRH), reachability for string rewriting systems (SR), and reachability for Post canonical systems in normal form ($\mathsf{PCS}_{\mathsf{nf}}$).

We reduce $\mathsf{Halt}_{\mathsf{SBTM}}$ to SRH and SRH to SR. We then combine the ideas from these two proofs to a proof $\mathsf{Halt}_{\mathsf{SBTM}} \preceq_m \mathsf{TSR} \preceq_m \mathsf{SR}$, and finally reduce SR to $\mathsf{PCS}_{\mathsf{nf}}$.

Publications Sections 20.1–20.3 are based on the following paper, all other parts are novel.

[71] Forster, Heiter, and Smolka. "Verification of PCP-Related Computational Reductions in Coq" *International Conference on Interactive Theorem Proving*. 2018.

20.1 Definition

Def. string rewriting system

We model strings over a finite alphabet by $\mathbb{L}\mathbb{N}$, rather than modelling finite alphabets directly. Rewriting rules are pairs of strings, and we define the type of **string rewriting system**s as a list of rewriting rules: $\mathsf{SRS}_X := \mathbb{L}(\mathbb{L}X \times \mathbb{L}X)$ and $\mathsf{SRS} := \mathsf{SRS}_{\mathbb{N}}$. Given $R : \mathsf{SRS}$, we define the symbols of R, $\mathsf{sym}\, R$, to be the list of all numbers occurring in a rule of R.

[1] Technically, "decidability" was not yet used as a term. Thue asked for "a method, where one can always calculate in a predictable number of operations" [194].

In this chapter, for strings x and y we write xy instead of $x +\!\!+ y$ for conciseness. We define string rewriting with two inductive predicates:

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

Def. SRS reachability problem

The SRS **reachability problem** is defined as

$$\mathsf{SR}(R : \mathsf{SRS}, x : \mathbb{LN}, y : \mathbb{LN}) := x \succ_R^* y.$$

Def. generalised SRS halting problem

The **generalised** SRS **halting problem** asks for the reachability of a string $y_1 a_0 y_n$ for a given symbol a_0 from a string x via a SRS R:

$$\mathsf{SRH}(R : \mathsf{SRS}, x : \mathbb{LN}, a_0 : \mathbb{N}) := \exists y.\, a_0 \in y \land x \succ_R^* y$$

Def. Thue system reachability problem

Thue systems are rewriting systems where all rules can be used in both directions. Instead of defining Thue systems as $R : \mathsf{SRS}$ with the extra property that $\forall (u,v) \in R.\, (v,u) \in R$, we directly use SRS to model the **Thue system reachability problem** using the operations $\overleftrightarrow{R} := R +\!\!+ \overleftarrow{R}$ and $\overleftarrow{R} := [(v,u) \mid (u,v) \in R]$.

$$\mathsf{TSR}(R : \mathsf{SRS}, x : \mathbb{LN}, y : \mathbb{LN}) := x \succ_{\overleftrightarrow{R}}^* y$$

Lastly, Post canonical systems in normal form differ from string rewriting in that a rule (u,v) only applies to strings ux, which are rewritten to xv:

$$\frac{(u,v) \in R}{ux \supset_R xv} \qquad \frac{}{x \supset_R^* x} \qquad \frac{x \supset y \qquad y \supset_R^* z}{x \supset_R^* z}$$

Def. reachability problem for Post canonical systems in normal form

The **reachability problem for Post canonical systems in normal form** is then defined as

$$\mathsf{PCS}_{\mathsf{nf}}(R : \mathsf{SRS}, x : \mathbb{LN}, y : \mathbb{LN}) := x \supset_R^* y.$$

20.2 Reducing $\mathsf{Halt}_{\mathsf{SBTM}}$ to SRH

To reduce SBTM halting to SRH we encode instantaneous configurations (q,t) where q is a state and t a tape as strings $\overline{(q,t)}$ and introduce rewriting rules according to the transitions of the machine. This proof is a simplification of the undecidability proof for TSR by Post [191].

To define the encoding of tapes, let a simple binary Turing machine M with $|M|$ states be given. We fix symbols $\llcorner\!\lrcorner$, $\langle\!|$, $|\!\rangle$, tt and ff, and with $n_0, \dots, n_{|M|}$ fix $\mathsf{S}\,|M|$ many symbols corresponding to the states of M, all mutually different.

We define the encoding of a configuration $\overline{(q,t)}$ consisting of a state q and a tape (ls, c, rs) as the string $\langle\!| \overline{(\mathsf{rev}\, ls)}\; \overline{q}\; \overline{c}\; \overline{rs} \,|\!\rangle$, where $\overline{c}$ is defined as $\overline{\mathsf{true}} := \mathsf{tt}$, $\overline{\mathsf{false}} := \mathsf{ff}$, $\overline{\mathsf{None}} := \llcorner\!\lrcorner$, $\overline{\mathsf{Some}\, c} := \overline{c}$, and $\overline{i : \mathbb{F}_{\mathsf{S}\,|M|}} := n_i$. The encoding function is extended to lists by pointwise application.

We now define a string rewriting system R corresponding to M in Figure 20.1 and can immediately prove the forward direction of a simulation:

Fact 20.1. If $M(q,t) \triangleright (q',t')$, then $\overline{(q,t)} \succ_R^* \overline{(q',t')}$.

The reverse direction, we first need two auxiliary results:

Read	*Write*	*Move*	u	v	u	v
None	None	L	$(\!\vert \overline{q_1}\,␣$	$(\!\vert \overline{q_2}\,␣$	$l\,\overline{q_1}\,␣$	$\overline{q_2}\,l$
None	None	N	$\overline{q_1}\,␣$	$\overline{q_2}\,␣$		
None	None	R	$\overline{q_1}\,␣\vert\!)$	$\overline{q_2}\,␣\vert\!)$	$\overline{q_1}\,␣\,r$	$\overline{q_2}\,r$
None	Some b	L	$(\!\vert \overline{q_1}\,␣$	$(\!\vert \overline{q_2}\,␣\,\overline{b}$	$l\,\overline{q_1}\,␣$	$\overline{q_2}\,l\,\overline{b}$
None	Some b	N	$\overline{q_1}␣$	$\overline{q_2}\,\overline{b}$		
None	Some b	R	$\overline{q_1}\,␣\vert\!)$	$b\,\overline{q_2}\,␣\vert\!)$	$\overline{q_1}\,␣\,r$	$b\,\overline{q_2}\,r$
Some a	None	L	$(\!\vert \overline{q_1}\,a$	$(\!\vert \overline{q_2}\,␣\,\overline{a}$	$l\,\overline{q_1}\,\overline{a}$	$\overline{q_2}\,l\,\overline{a}$
Some a	None	N	$\overline{q_1}\,\overline{a}$	$\overline{q_2}\,\overline{a}$		
Some a	None	R	$\overline{q_1}\,\overline{a}\vert\!)$	$a\,\overline{q_2}\,␣\vert\!)$	$\overline{q_1}\,\overline{a}\,r$	$a\,\overline{q_2}\,r$
Some a	Some b	L	$(\!\vert \overline{q_1}\,a$	$(\!\vert \overline{q_2}\,␣\,\overline{b}$	$l\,\overline{q_1}\,\overline{a}$	$\overline{q_2}\,l\,\overline{b}$
Some a	Some b	N	$\overline{q_1}\,\overline{a}$	$\overline{q_2}\,c$		
Some a	Some b	R	$\overline{q_1}\,\overline{a}\vert\!)$	$\overline{b}\,\overline{q_2}\,␣\vert\!)$	$\overline{q_1}\,\overline{a}\,r$	$\overline{b}\,\overline{q_2}\,r$

Figure 20.1.: Rewriting rules $(u, v) \in R$ if $\delta(q_1, c) = (q_2, w, m)$, where c is in the column *Read*, w in the column *Write*, and m in the column *Move*.

Fact 20.2. If $x\,\overline{q}\,y = \overline{(q', (ls, crs))}$, then $x = (\!\vert\, \overline{(\mathsf{rev}\, ls)}$, $q = q'$, and $y = \overline{c}\,\overline{rs}\vert\!)$.

This suffices to prove the following inversion lemma:

Fact 20.3. If $\overline{(q, (ls, c, rs))} \succ_R z$, then $\delta_M(q, c) = \mathsf{Some}\,(q', w, m)$ for some q', w, m, and $z = \overline{(\mathsf{mv}\, m\, (\mathsf{wr}\, w\, (ls, c, rs)))}$.

To prove the reduction theorem reducing $\mathsf{Halt}_{\mathsf{SBTM}}$ to SRH we need to be able to pick a single symbol indicating halting, but in general a single tape binary Turing machine can have more than one halting state. We thus first introduce the halting problem for single tape binary Turing machines with a unique halting state $\mathsf{Halt}_{\mathsf{SBTMu}}$:

$$\mathsf{Halt}_{\mathsf{SBTMu}}(M : \mathsf{SBTM}, t : \mathbb{LB} \times \mathbb{OB} \times \mathbb{LB}, q : \mathbb{F}_{\mathsf{S}\,|M|}, H : \mathsf{uhalt}\, M\, q) := \exists t'.\ M(0, t) \triangleright (q, t')$$

where $\mathsf{uhalt}\, M\, q := \forall c.\ \delta_M(q, c) = \mathsf{None} \wedge \forall q'.\ \delta_M(q', c) = \mathsf{None} \to q = q'$.

Lemma 20.4. $\mathsf{Halt}_{\mathsf{SBTM}} \preceq_m \mathsf{Halt}_{\mathsf{SBTMu}}$

Proof. Given M construct M' with exactly one more state q_h, $\delta_{M'}(q_h, c) = \mathsf{None}$ and whenever $\delta_M(q, c) = \mathsf{None}$, $\delta_{M'}(q, c) = \mathsf{Some}\,(q_h, \mathsf{None}, \mathsf{N})$. ∎

Theorem 20.5. $\mathsf{Halt}_{\mathsf{SBTMu}} \preceq_m \mathsf{SRH}$ and thus $\mathcal{U}(\mathsf{SRH})$.

Proof. We prove $\mathsf{Halt}_{\mathsf{SBTMu}}(M, t, q, H) \leftrightarrow \mathsf{SRH}(R, \overline{(0, t)}, \overline{q})$. The forward direction is immediate from Fact 20.1. The reverse direction follows by induction on $\succ_R^*$, Fact 20.3, and exploiting that q is the unique halting state. ∎

20.3 Reducing SRH to SR

To reduce SRH to SR, fix $R : \mathsf{SRS}$, a string x and a symbol a_0 and define:

$$R' := R +\!\!+ [(a_0 a, a_0) \mid a \in \Sigma] +\!\!+ [(a a_0, a_0) \mid a \in \Sigma] \text{ with } \Sigma := \mathsf{sym}\, R +\!\!+ x +\!\!+ [a_0]$$

That is, R' has the same rules as R, but once a_0 is reached via R (and $\mathsf{SRH}(R, x, a_0)$ holds), all symbols apart from a_0 are deleted. The idea of using deletion rules is also present in the TSR undecidability proof by Post [191].

Lemma 20.6. If $a_0 \in x$, then $x \succ_{R'}^* a_0$.

Proof. If $a_0 \in x$ then $x = x_1 a_0 x_2$ for some x_1, x_2. We first prove $x_1 a_0 x_2 \succ_{R'}^* x_1 a_0$ by induction on x_2, and then $x_1 a_0 \succ_{R'}^* a_0$ by induction on x_1. ∎

Theorem 20.7. $\mathsf{SRH} \preceq_m \mathsf{SR}$ and thus $\mathcal{U}(\mathsf{SR})$.

Proof. We show $\mathsf{SRH}(R, x, a_0) \leftrightarrow \mathsf{SR}(R', x, a_0)$. The direction from right to left is a straightforward induction on $\succ_R^*$. For the other direction, the claim then follows by induction on $\succ_{R'}^*$ using Lemma 20.6. ∎

20.4 Reducing $\mathsf{Halt}_{\mathsf{SBTM}}$ to TSR

The proof ideas from Sections 20.2 and 20.3 are essentially a decomposition of the undecidability proof of TSR by Post [191]. We now re-compose the ideas and extend the rewriting system R for a given SBTM with carefully chosen deletion rules to prove the undecidability for TSR. Note that we cannot simply prove $\mathsf{Halt}_{\mathsf{SBTM}} \preceq_m \mathsf{SR} \preceq_m \mathsf{TSR}$ to obtain the undecidability of TSR: Post's proof idea only applies to rewriting systems in a very particular shape, which is fulfilled by the rewriting system corresponding to a SBTM. Transforming arbitrary SRS into such a shape would likely be harder than starting the undecidability proof at $\mathsf{Halt}_{\mathsf{SBTM}}$.

Let again a SBTM M be given, with starting state q_0 and unique halting state q_h. We reuse the system R constructed in Section 20.2, and add some carefully chosen deletion rules D, similar to the reduction from SRH to SR. Let $\boxdot$ be a fresh symbol not occurring in R. We define:

$$\begin{aligned} D := &\big[(q_h c a, q_h c) \mid c \in [\mathsf{tt}, \mathsf{ff}, \text{␣}], a \in [\mathsf{tt}, \mathsf{ff}]\big] +\!\!+ \\ &\big[(a q_h, q_h) \mid a \in [\mathsf{tt}, \mathsf{ff}]\big] +\!\!+ \\ &\big[(\langle\!| q_h c |\!\rangle, \boxdot) \mid a \in [\mathsf{tt}, \mathsf{ff}]\big] \end{aligned}$$

Note that the rules only delete symbols ␣, tt, and ff to the right and left of a halting state, no tape delimiters $\langle\!|$ and $|\!\rangle$ and no state symbols q_i. In particular, that means that the tape delimiters $\langle\!|$ and $|\!\rangle$ stay in the string until the very end, where they are replaced by $\boxdot$. Additionally, by not adding deletion rules for symbols q_i we avoid that $\overleftrightarrow{D}$ contains rules *introducing* a second state q_i in a string, which then again would be subject to forward applications of rules from R.

In the system $R +\!\!+ D$ a SBTM run

$$(q_0, t_0) \rightsquigarrow (q_1, t_1) \rightsquigarrow \cdots \rightsquigarrow (q_h, t_h)$$

can be simulated with a rewriting sequence

$$\overline{(q_0, t_0)} \succ_R \overline{(q_1, t_1)} \succ_R \ldots \succ_R \overline{(q_h, t_h)} \succ_D^* \langle\!| q_h c |\!\rangle \succ_D \boxdot$$

The following fact then suffices for the forward direction of the reduction:

Fact 20.8. $\overline{(q_h, t)} \succ_{\overleftrightarrow{R \uplus D}} \boxdot$

Corollary 20.9. If $\mathsf{Halt}_{\mathsf{SBTM}}(M, t)$, then $\overline{(q_0, t)} \succ_{\overleftrightarrow{R \uplus D}} \boxdot$.

The other direction is considerably more intricate. We have to prove that using rules from $\overleftarrow{R}$, D, or $\overleftarrow{D}$ in a rewriting sequence $\overline{(q_0, t)} \succ^*_{\overleftrightarrow{R \uplus D}} \boxdot$ still allows to deduce $\mathsf{Halt}_{\mathsf{SBTM}}(M, t)$.

Similar to Fact 20.3 we prove the following inversions, which suffice for the reduction theorem.

Fact 20.10. The following hold:

1. If $\overline{(q, t)} \succ_{\overleftarrow{R}} z$, then there are q', ls, c, rs, w, and m such that $\delta_M(q', c) = \mathsf{Some}(q, w, m)$, $t = \mathsf{mv}\, m\, (\mathsf{wr}\, w\, t')$ and $z = \overline{(q', t')}$.
2. If $\overline{(q, t)} \succ_D z$, then $q = q_h$ and either $z = \boxdot$ or $z = \overline{(q, t')}$ for some tape t'.
3. If $\overline{(q, t)} \succ_{\overleftarrow{D}} z$, then $q = q_h$ and $z = \overline{(q, t')}$ for some tape t'.

Theorem 20.11. $\mathsf{Halt}_{\mathsf{SBTMu}} \preceq_m \mathsf{TSR}$ and thus $\mathcal{U}(\mathsf{TSR})$.

Proof. The forward direction is by Corollary 20.9. The backwards direction follows by induction on $\succ^*_{\overleftrightarrow{R \uplus D}}$ via Fact 20.10. ∎

20.5 Reducing SR to $\mathsf{PCS}_{\mathsf{nf}}$

Let $R : \mathsf{SRS}$, $x_0, y_0 : \mathbb{LN}$, $\Sigma := x_0 +\!\!+ y_0 +\!\!+ \mathsf{sym}\, R$ and let $\#$ be fresh for Σ.

We define

$$R' := R +\!\!+ [(a, a) \mid a \in \Sigma].$$

Using the copy rules (a, a) we can prove:

Fact 20.12. If $x \subseteq \Sigma$ then $xy \supset^*_{R'} yx$.

The forward direction of the simulation is then straightforward.

Fact 20.13. If $x \subseteq \Sigma$, $y \subseteq \Sigma$ and $x \succ^*_R y$, then $x\# \supset^*_{R'} y\#$

The backwards direction is more involved. We prove the following generalised inductive invariant:

Lemma 20.14. If $x_1, x_2, y_1, y_2 \subseteq \Sigma$ and $x_2\#x_1 \supset^*_{R'} y_2\#y_1$, then $x_1x_2 \succ^*_R y_1y_2$.

Proof. By induction on $\supset^*_{R'}$ with x_1, x_2, y_1, y_2 generalised. ∎

Theorem 20.15. $\mathsf{SR} \preceq_m \mathsf{PCS}_{\mathsf{nf}}$ and thus $\mathcal{U}(\mathsf{PCS}_{\mathsf{nf}})$

Proof. We prove $x \succ^*_R y \leftrightarrow x\# \supset^*_{R'} y\#$. The forward direction is immediate by Fact 20.13. The backwards direction follows from Lemma 20.14 by picking $x_1 = y_1 = [\,]$. ∎

CHAPTER 21

The Post correspondence problem

The Post correspondence problem PCP [190] is often used as seed for undecidability results due to its simplicity. It is considered part of the curriculum in computability theory and covered in many textbooks. Various undecidability proofs for PCP are in the literature:

- by a direct reduction $\mathsf{Halt}_{\mathsf{TM}^1} \preceq_m \mathsf{PCP}$ [207],
- via modified PCP (MPCP) proving $\mathsf{Halt}_{\mathsf{TM}^1} \preceq_m \mathsf{MPCP} \preceq_m \mathsf{PCP}$ [113],
- via SR proving $\mathsf{Halt}_{\mathsf{TM}} \preceq_m \mathsf{SR} \preceq_m \mathsf{PCP}$ [48].

In his seminal paper, Post himself gives a reduction $\mathsf{PCS}_{\mathsf{nf}} \preceq_m \mathsf{PCP}$ [190].

We here factor the undecidability proof of PCP as reduction $\mathsf{SR} \preceq_m \mathsf{MPCP} \preceq_m \mathsf{PCP}$, obtaining simple proofs with clear inductive invariants. Note that while a reduction $\mathsf{PCS}_{\mathsf{nf}} \preceq_m \mathsf{MPCP}$ is possible (and simpler than Post's reduction $\mathsf{PCS}_{\mathsf{nf}} \preceq_m \mathsf{PCP}$), it seems to be not simpler than $\mathsf{SR} \preceq_m \mathsf{MPCP}$.

In Section 21.1 we define PCP, MPCP and a boolean version, BPCP. In Section 21.2 we prove the undecidability of MPCP, in Section 21.3 we reduce MPCP to PCP and in Section 21.4 PCP to BPCP.

[190] Post. 1946. A variant of a recursively unsolvable problem.

[207] Sipser. 2006. Introduction to the Theory of Computation.

[113] Hopcroft, Motwani, and Ullman. 2006. Introduction to Automata Theory, Languages, and Computation (3rd Edition).

[48] Davis and Weyuker. 1983. Computability, complexity, and languages - fundamentals of theoretical computer science.

[190] Post. 1946. A variant of a recursively unsolvable problem.

Publications The chapter is mainly based on [71]. The inductive definition of PCP was introduced in [73], and Section 21.4 is based on [81].

[71] Forster, Heiter, and Smolka. "Verification of PCP-Related Computational Reductions in Coq." *International Conference on Interactive Theorem Proving*. 2018.

[73] Forster, Kirst, and Smolka. "On Synthetic Undecidability in Coq, with an Application to the Entscheidungsproblem." *Proceedings of the 7th ACM SIGPLAN Conference on Certified Programs and Proofs*. 2019.

[81] Forster and Larchey-Wendling. "Certified Undecidability of Intuitionistic Linear Logic via Binary Stack Machines and Minsky Machines." *Proceedings of the 7th ACM SIGPLAN Conference on Certified Programs and Proofs*. 2019.

21.1 Definition

We give several equivalent definitions of the Post correspondence problem, suitable for different reductions. In all cases, the Post correspondence problem is defined over a set of cards containing a top and a bottom string over a type X. Cards are modelled as $c : \mathbb{L}X \times \mathbb{L}X$, and sets of cards as $R : \mathbb{L}(\mathbb{L}X \times \mathbb{L}X)$.

We define the post correspondence problem PCP_X using a relation $R \triangleright (x, y)$:

$$\frac{(u,v) \in R}{R \triangleright (u,v)} \qquad \frac{(u,v) \in R \qquad R \triangleright (x,y)}{R \triangleright (u \mathbin{+\!\!+} x, v \mathbin{+\!\!+} y)}$$

Def. Post correspondence problem

We then define the **Post correspondence problem** PCP and the **binary Post correspon-**

dence problem BPCP as follows:

$$\mathsf{PCP}_X(R : \mathsf{SRS}_X) := \exists x.\, R \triangleright (x, x)$$

$$\mathsf{PCP}(R : \mathsf{SRS}_{\mathbb{N}}) := \mathsf{PCP}_{\mathbb{N}} R \qquad \mathsf{BPCP}(R : \mathsf{SRS}_{\mathbb{B}}) := \mathsf{PCP}_{\mathbb{B}} R$$

While the modified Post correspondence problem can be defined using an adapted derivability relation, we refrain from doing so and immediately proceed to a definition based on stacks, which is more suitable to define MPCP.

21.1.1 Definition using stacks

Def. stack (of cards)

We call a list $A : \mathbb{L}(\mathbb{L}X \times \mathbb{L}X)$ a **stack**. We define the first and second trace of stacks as

$$\tau_1[\,] := [\,] \qquad \tau_2[\,] := [\,]$$

$$\tau_1((x, y) :: A) := x +\!\!+ \tau_1 A \qquad \tau_2((x, y) :: A) := y +\!\!+ \tau_2 A$$

We can then define PCP' as

$$\mathsf{PCP}'_X(R : \mathsf{SRS}_X) := \exists A \subseteq R.\, A \neq [\,] \wedge \tau_1 A = \tau_2 A$$

Def. modified Post correspondence problem

and the **modified Post correspondence problem** MPCP as

$$\mathsf{MPCP}(R : \mathsf{SRS}_{\mathbb{N}}, (x, y) : \mathbb{L}\mathbb{N} \times \mathbb{L}\mathbb{N}) := \exists A \subseteq (x, y) :: R.x +\!\!+ \tau_1 A = y +\!\!+ \tau_2 A$$

Fact 21.1. $\mathsf{PCP}_X R \leftrightarrow \mathsf{PCP}'_X R$

21.1.2 Definition using indices

The most commonly found definition of PCP in the literature, and also the definition used by Post [190] is based on natural number indices. Given $I : \mathbb{L}\mathbb{N}$ and $R : \mathsf{SRS}_X$, we define the traces of I as:

$$\tau_1\, R\, [\,] := [\,] \qquad \tau_2\, R\, [\,] := [\,]$$

$$\tau_1\, R\, (i :: I) := x +\!\!+ \tau_1\, R\, I \qquad \tau_2\, R\, (i :: I) := y +\!\!+ \tau_2\, R\, I \qquad (\text{if } R[i] = \mathsf{Some}\,(x, y))$$

$$\tau_1\, R\, (i :: I) := \tau_1\, R\, I \qquad \tau_2\, R\, (i :: I) := \tau_2\, R\, I \qquad (\text{if } R[i] = \mathsf{None})$$

We then define $\mathsf{PCP}_{\mathsf{i}} : \mathsf{SRS}_X \to \mathbb{P}$ and $\mathsf{MPCP}_{\mathsf{i}} : \mathsf{SRS}_X \times (\mathbb{L}X \times \mathbb{L}X) \to \mathbb{P}$ based on indices:

$$\mathsf{PCP}_{\mathsf{i},X}(R) := \exists I : \mathbb{L}\mathbb{N}.\, (\forall i \in I.\, i < |R|) \wedge I \neq [\,] \wedge \tau_1\, R\, I = \tau_2\, R\, I$$

$$\mathsf{MPCP}_{\mathsf{i},X}(R, (x, y)) := \exists I : \mathbb{L}\mathbb{N}.\, (\forall i \in I.\, i \leq |R|) \wedge x +\!\!+ \tau_1\, ((x, y) :: R)\, I = \tau_2\, ((x, y) :: R)\, I$$

$$\mathsf{PCP}_{\mathsf{i}}(R : \mathsf{SRS}_{\mathbb{N}}) := \mathsf{PCP}_{\mathsf{i},\mathbb{N}} R \qquad \mathsf{BPCP}_{\mathsf{i}}(R : \mathsf{SRS}_{\mathbb{B}}) := \mathsf{PCP}_{\mathsf{i},\mathbb{B}} R$$

Fact 21.2. $\mathsf{PCP}_X R \leftrightarrow \mathsf{PCP}_{\mathsf{i},X} R$ and $\mathsf{MPCP}_X(R, (x, y)) \leftrightarrow \mathsf{MPCP}_{\mathsf{i},X}(R, (x, y))$.

21.2 Reducing SR to MPCP

Let R : SRS, $x_0 : \mathbb{LN}$, and $y_0 : \mathbb{LN}$ be given. Let $\Sigma := x_0 +\!\!+ y_0 +\!\!+ \mathsf{sym}R$ and let # and \$ be different and both fresh for Σ. We define

$$d := (\$, \$x_0\#)$$

$$e := (y_0\#\$, \$)$$

$$R' := d :: e :: R +\!\!+ (\#, \#) :: [(a, a) \mid a \in \Sigma]$$

The idea of the reduction is as follows: Assume $R = [(bc, a), (aa, b)]$ and $x_0 = abc \succ_R aa \succ_R b = y_0$. We then have $\Sigma = [a, b, c]$, $d = (\$, \$abc\#)$, $e = (b\#\$, \$)$, and $R' = [d, e, (bc, a), (aa, b), (a, a), (b, b), (c, c)]$. Written suggestively, the following stack mirroring the rewriting sequence from above has matching traces:

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

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

For the general reduction we will use MPCP as target. We first prove the inductive invariant for the forward direction:

Fact 21.3. Let $x \subseteq \Sigma$ and $x \succ_R^* y_0$. Then $\exists A \subseteq R'.\ \tau_1 A = x\#\tau_2 A$.

For the other direction we again need a more involved inductive invariant:

Fact 21.4. Let $x, y \subseteq \Sigma$, $A \subseteq R$, and $\tau_1 A = x\#y\tau_2 A$. Then $yx \succ_R^* y_0$.

Theorem 21.5. SR $\preceq_m$ MPCP and thus $\mathcal{U}$(MPCP).

Proof. We show $x_0 \succ_R^* y_0 \leftrightarrow \mathsf{MPCP}(R', (x_0, y_0))$. The direction from left to right is by Fact 21.3, the other direction follows from Fact 21.4 with $y = [\,]$. ∎

21.3 Reducing MPCP to PCP

We reduce MPCP to PCP following the proof by Hopcroft, Motwani, and Ullman [113].

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

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

if and only if we have

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

This idea is already present in Post's work [190] as a component of the reduction from $\mathsf{PCS}_{\mathsf{nf}}$ to PCP.

Now let $R : \mathsf{SRS}$, $x_0 : \mathbb{LN}$, and $y_0 : \mathbb{LN}$ be given. Let $\Sigma := x_0 +\!\!+ y_0 +\!\!+ \mathsf{sym}\, R$ and let $\#$ and \$ be distinct and both fresh for Σ.

We define two interleaving functions ${}^{\#}x$ and $x^{\#}$ as follows:

$$ {}^{\#}[] := [] \qquad\qquad [\,]^{\#} := [] $$

$$ {}^{\#}(a :: x) := \# :: a :: +\!\!+({}^{\#}x) \qquad\qquad (a :: x)^{\#} := a :: \# :: (x^{\#}) $$

We then define:

$$ d := (\$({}^{\#}x_0)\,,\, \$\#(y_0^{\#})) $$

$$ e := (\#\$\,,\, \$) $$

$$ R' := d :: e :: [({}^{\#}x\,,\, y^{\#}) \mid (x, y) \in (x_0, y_0) :: R \wedge (x/y) \neq ([\,],[\,])] $$

The inductive invariant for the forward direction of the reduction is as follows:

Fact 21.6. Let $A \subseteq (x_0, y_0) :: R$. If $x +\!\!+ \tau_1 A = y +\!\!+ \tau_2 A$, then $\exists B \subseteq P.\ {}^{\#}x +\!\!+ \tau_1 B = \# :: y^{\#} +\!\!+ \tau_2 B$.

The backwards direction reads similarly.

Fact 21.7. Let $B \subseteq R'$ and $x, y \subseteq \Sigma$. If ${}^{\#}x +\!\!+ \tau_1 B = \# :: y^{\#} +\!\!+ \tau_2 B$, then $\exists A \subseteq (x_0, y_0) :: R.\ x +\!\!+ \tau_1 A = y +\!\!+ \tau_2 A$.

Theorem 21.8. $\mathsf{MPCP} \preceq_m \mathsf{PCP}$ and thus $\mathcal{U}(\mathsf{PCP})$.

Proof. We prove $\mathsf{MPCP}(R, (x_0, y_0)) \leftrightarrow \mathsf{PCP}' R$. The direction from left to right is by Fact 21.6, the other direction by Fact 21.7. ∎

21.4 Reducing PCP to BPCP

We define a function $f : \mathbb{LN}^* \to \mathbb{LB}^*$ encoding natural number strings as booleans strings as

$$ f\,[\,] := [\,] \qquad\qquad f\,(n :: x) := \mathsf{true}^n +\!\!+ \mathsf{false} :: f\,x $$

and we extend f to cards and stacks by pointwise application.

We define the inverse function $g : \mathbb{LB} \to \mathbb{N} \to \mathbb{LN}$ with an auxiliary argument by

$$ g\ [\,]\ n := [\,] \qquad g\ (\mathsf{true} :: x)\ n := g\ x\ (1 + n) \qquad g\ (\mathsf{false} :: x)\ n := n :: g\ x\ 0 $$

We define $g\ x := g\ x\ 0$ and again extend it pointwise to cards and stacks.

Fact 21.9. $g(f\,x) = x$

Theorem 21.10. $\mathsf{PCP} \preceq_m \mathsf{BPCP}$ and thus $\mathcal{U}(\mathsf{BPCP})$.

Proof. We prove that $\mathsf{PCP}\, R \leftrightarrow \mathsf{BPCP}(f\, R)$. The direction from left to right is easy. The other direction uses Fact 21.9. ∎

CHAPTER 22

Context-free grammars

Context-free grammars were introduced by Chomsky [29] to describe the structure of natural language. Their use in computer science was popularised by the Algol programming language, and context-free grammars still form the main tool to describe the syntax of programming languages.

[29] Chomsky. 1956. Three models for the description of language.

The central problem concerning context-free grammars is the word problem, which is decidable. In 1961 Bar-Hillel, Perles, and Shamir [7] proved that the problem CFI of determining whether two context-free grammars intersect is undecidable, by reduction from PCP. Hesselink [109] factors the proof into a reduction from PCP to the problem CFP of determining whether a context-free grammar contains a palindrome. Since the grammar of all palindromes is context-free, it is straightforward to deduce a reduction from CFP to CFI.

[7] Bar-Hillel, Perles, and Shamir. 1961. On formal properties of simple phrase structure grammars.

[109] Hesselink. 2015. Post's Correspondence Problem and the Undecidability of Context-Free Intersection.

We here use Hesselink's idea, but factor the proof via Post grammars, a special case of context-free grammars tailored for reductions from PCP.

Publication This chapter is based on:

[71] Forster, Heiter, and Smolka. "Verification of PCP-Related Computational Reductions in Coq" *International Conference on Interactive Theorem Proving*. 2018.

22.1 Definition

Context-free grammars are a restricted form of string rewriting systems, namely such where the left-hand side of a rule only consists of one (non-terminal) symbol. Conventionally, one separates the alphabet into terminal symbols (which cannot appear on the left hand side of rules) and non-terminal symbols. Different to string rewriting system, context-free grammars have a (non-terminal) start symbol.

We model context-free grammars as pairs of start symbols and rules: $\mathsf{CFG} := \mathbb{N} \times \mathbb{L}(\mathbb{N} \times \mathbb{L}\mathbb{N})$. We do not formally distinguish terminal and non-terminal symbols, but rather consider every symbol occurring on the left-hand side of a rule as non-terminal and all others as terminal.

We reuse the string rewriting relation from Chapter 20 to define the language of a context-free grammar:

$$x \in \mathcal{L}_{\mathsf{CFG}}(a, G) := a \succ^*_{\hat{G}} x \wedge \nexists y. x \succ_{\hat{G}} y \text{ where } \hat{G} := [([l], r) \mid (l, r) \in G]$$

A string x is a palindrome if $x = \mathsf{rev}\, x$. The **context-free palindrome** problem CFP is defined as

Def. context-free palindrome

$$\mathsf{CFP}(G : \mathsf{CFG}) := \exists x \in \mathcal{L}_{\mathsf{CFG}} G.\ x = \mathsf{rev}\, x.$$

The **context-free intersection** problem CFI is defined as

Def. context-free intersection

$$\mathsf{CFI}(G_1 : \mathsf{CFG}, G_2 : \mathsf{CFG}) := \exists x.\ x \in \mathcal{L}_{\mathsf{CFG}}(G_1) \wedge x \in \mathcal{L}_{\mathsf{CFG}}(G_2).$$

Post grammars are special context-free grammars with a single non-terminal symbol S where all rules are of the form $S \succ xSy$ or $S \succ xay$ for a fixed symbol a. We model Post grammars as pairs $\mathsf{PG} := \mathbb{N} \times \mathsf{SRS}$. Given a Post grammar (a, R) we model derivations as stacks $A \subseteq R$ and define a projection function translating derivations to words:

$$\sigma_a[\,] := a \qquad\qquad \sigma_a((x,y) :: A) := x +\!\!+ \sigma_a A +\!\!+ y$$

The language of a Post grammar is defined via the projection function:

$$x \in \mathcal{L}_{\mathsf{PG}} := \exists A \subseteq R.\, A \neq [\,] \wedge \sigma_a A = x$$

Def. context-free Post grammar palindrome

The **context-free Post grammar palindrome** problem is defined as

$$\mathsf{CFPP}(a, R) := \exists x \in \mathcal{L}_{\mathsf{PG}}.\, x = \mathsf{rev}\, x.$$

22.2 Reducing PCP to CFPP

Notice that a stack $A = [(x_1, y_1), \ldots, (x_n, y_n)]$ matches if and only if the string

$$x_1 \ldots x_n \# \mathsf{rev}\, y_n \ldots \mathsf{rev}\, y_1$$

is a palindrome due to the following:

Fact 22.1. Let $\#$ be fresh for x and y. Then $x +\!\!+ \# +\!\!+ y$ is a palindrome if and only if $y = \mathsf{rev}\, x$.

Provided an SRS R we prove that PCP R holds if and only if the Post grammar $(\#, \gamma R)$ contains a palindrome, with

$$\gamma A := [(x, \mathsf{rev}\, y) \mid (x, y) \in A.]$$

Lemma 22.2. If $\#$ is fresh for A then the following hold:

1. $\sigma_\#(\gamma A) = \tau_1 A +\!\!+ \# +\!\!+ \mathsf{rev}\,(\tau_2 A)$
2. $\tau_1 A = \tau_2 A$ if and only if $\sigma_\#(\gamma A)$ is a palindrome.
3. $\gamma(\gamma A) = A$
4. $A \subseteq \gamma B \to \gamma A \subseteq B$

Proof. All are by induction on A. The proof of (2) needs (1) and Fact 22.1 ∎

Theorem 22.3. PCP $\preceq_m$ CFPP and thus $\mathcal{U}(\mathsf{CFPP})$.

Proof. Let R be a string rewriting system and $\#$ fresh for R. We prove $\mathsf{PCP}R \leftrightarrow \mathsf{CFP}(\#, \gamma R)$.

The direction from left to right follows with (2), (3), and (4).

For the direction from right to left let $[\,] \neq B \subseteq \gamma R$ s.t. $\sigma_\# B$ is a palindrome. By (3) and (4) we have $\gamma B \subseteq R$. Since also $B = \gamma(\gamma B)$ by (3) and (4) we have $\tau_1(\gamma B) = \tau_2(\gamma B)$ by (2). ∎

22.3 Reducing CFPP to CFP

We reduce the word problem for context-free Post grammars to the word problem of context-free grammars, yielding immediately a reduction from CFPP to CFP.

Let a Post grammar (a, R) be given. Let S be fresh for a, R and

$$G := (S, (S, [S]) :: [(S, u +\!\!+ [S] +\!\!+ v) \mid (u, v) \in R] +\!\!+ [(S, u +\!\!+ [a] +\!\!+ v) \mid (u, v) \in R]).$$

We will prove that $x \in \mathcal{L}_{\mathsf{PG}}(a, R) \leftrightarrow x \in \mathcal{L}_{\mathsf{CFG}} G$.
The direction from left to right needs the following three properties:

Lemma 22.4. The following hold:

1. $S \notin x$ if and only if $\neg\exists y.\, x \succ_G y$.
2. If $S \succ_G^* x$ then S occurs at most once in x.
3. If $A \subseteq R$ then $A = [\,]$ or $S \succ_G^* \sigma_a A$.

Proof. (1) is proved by induction on x, (2) by induction on the derivation, and (3) by induction on A. ∎

The direction from right to left needs the following:

Fact 22.5. The following hold:

1. Let $\sigma_b A = x +\!\!+ [c] +\!\!+ y$ and c not occur in x, y, or A. Then $b = c$.
2. Let $\sigma_b A = x +\!\!+ [b] +\!\!+ y$ and b not occur in x, y. Then $\sigma_c(A +\!\!+ [(u, v)]) = x +\!\!+ u +\!\!+ [c] +\!\!+ v +\!\!+ y$.

Fact 22.6. $\mathcal{L}_{\mathsf{PG}} \preceq_m \mathcal{L}_{\mathsf{CFG}}$

Theorem 22.7. CFPP $\preceq_m$ CFP and thus $\mathcal{U}(\mathsf{CFP})$.

22.4 Reducing CFP to CFI

The language of palindromes is context-free:

Lemma 22.8. Let $\Sigma : \mathbb{LN}$ be given. There is G such that for all x with $x \subseteq \Sigma$, $x \in \mathcal{L}_{\mathsf{CFG}} G$ if and only if $x = \mathsf{rev}\, x$.

Proof. Let Σ be given and S be fresh for Σ. Define $G := (S, (S, [\,]) :: [(S, c) \mid c \in \Sigma] +\!\!+ [(S, cSc) \mid c \in \Sigma])$.

For the direction from left to right we prove that if $x \subseteq S :: \Sigma$, $x = \mathsf{rev}\, x$, and S occurs at most once in x, then $S \succ_G^* x$ via the induction scheme

$$\forall X p.\, p[\,] \to (\forall x. p[x]) \to (\forall x_1 l x_2.\, pl \to p(x_1 :: l +\!\!+ [x_2])) \to \forall l. pl.$$

For the converse direction we prove that if $S \succ_G^* x$ then $x = \mathsf{rev}\, x$ and S occurs at most once in x by induction on the derivation. ∎

Theorem 22.9. CFP $\preceq_m$ CFI and thus $\mathcal{U}(\mathsf{CFI})$.

Proof. Let a grammar G' be given. Let Σ all symbols in G'.

Then $\mathsf{CFP}(S, G')$ if and only if $\mathsf{CFI}(G', G)$ with G from Lemma 22.8. ∎

CHAPTER 23

First-order logic

The *Entscheidungsproblem* was identified by Hilbert and Ackermann to be the "central problem of mathematical logic" in 1928 [110], and can be considered as one of the main driving forces in the development of computability. From a classical perspective, the Entscheidungsproblem can be equivalently phrased as asking for a decision procedure for the problem of determining whether a first-order formula

- is valid in all Tarski-models,
- is satisfiable in any Tarski-model, or
- is provable in classical natural deduction.

While they are constructively not equivalent, we prove all of these problems undecidable, and furthermore factor the proofs such that we also obtain undecidability proofs for the problems of determining whether a first-order formula

- is provable in intuitionistic natural deduction, or
- is provable in minimal natural deduction (i.e. intuitionistic deduction without explosion).

Other model-theoretic semantics for first-order logic such as Kripke, Heyting, or dialogue semantics can be similarly treated, but we do not cover them here.

Instead of using first-order formulas to encode Turing machines (as Turing did) or λ-terms (as Church did), we express the solvability of a BPCP instance as first order formula φ_R which is valid (or provable) if and only if BPCP R holds. We follow the proof of Manna [164], who attributes the proof idea to Floyd. We encode lists of booleans $[b_1, \ldots, b_n]$ as terms $f_{b_1}(\ldots(f_{b_n} e))$ and use Horn clauses to express the inductive rules of BPCP introduced in Section 21.1.

Furthermore, the proof for classical natural deduction relies crucially on a combination of the Gödel-Gentzen double negation translation [92, 89] and Friedman's A-translation [87].

[110] Hilbert and Ackermann. 1928. Grundzüge der theoretischen Logik.

[92] Gödel. 1930. Die Vollständigkeit der Axiome des logischen Funktionenkalküls.

[89] Gentzen. 1936. Die Widerspruchsfreiheit der reinen Zahlentheorie.

[87] Friedman. 1978. Classically and intuitionistically provably recursive functions.

Publications This chapter is based on

[73] Forster, Kirst, and Smolka. "On synthetic undecidability in Coq, with an application to the Entscheidungsproblem." *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 2019.

23.1 Definition

We fix the term signature to contain a constant symbol e and two unary functions f_0 and f_1:

$$t : \mathsf{tm} ::= x \mid e \mid f_0 t \mid f_1 t \quad \textit{where } x : \mathbb{N}$$

We consider the $\forall\rightarrow$ fragment of first-order logic with falsity, one propositional constant Q and one binary predicate P:

$$\varphi, \psi : \mathsf{fm} ::= \dot{\bot} \mid Q \mid P t_1 t_2 \mid \varphi \dot{\rightarrow} \psi \mid \forall \varphi$$

The current Coq mechanisation as part of the Coq library of Undecidability Proofs contains two formalisations of this proof. One [73] due to the author of this thesis using names, and

one parametric in operations, quantifiers, term-, and predicate signatures by Kirst [128]. A previous now deprecated formalisation [74] by the author was based on the Autosubst 2 tool [218, 217].

[218] Stark et al.. 2019. Autosubst 2: reasoning with multi-sorted de Bruijn terms and vector substitutions.

[217] Stark. 2019. Mechanising syntax with binders in Coq.

Given a (naive Tarski-style) model $\mathcal{M}$ for first-order logic consisting of a domain $D : \mathbb{T}$, term interpretations $[\![e]\!]_{\mathcal{M}} : D$, $[\![f_0]\!]_{\mathcal{M}} : D \to D$ and $[\![f_1]\!]_{\mathcal{M}} : D \to D$, and propositional interpretations $[\![Q]\!]_{\mathcal{M}} : \mathbb{P}$ and $[\![P]\!]_{\mathcal{M}} : D \to D \to \mathbb{P}$ and also given an environment $\rho : \mathbb{N} \to D$, we define the satisfaction relation $\vDash$ as

$$\hat{\rho} x := \rho x \qquad \hat{\rho} e := [\![e]\!] \qquad \hat{\rho}(f_0 t) := [\![f_0]\!](\hat{\rho} t) \qquad \hat{\rho}(f_1 t) := [\![f_1]\!](\hat{\rho} t)$$

$$\mathcal{M} \vDash_\rho \dot{\bot} := \bot \qquad \mathcal{M} \vDash_\rho Q := [\![Q]\!] \qquad \mathcal{M} \vDash_\rho P t_1 t_2 := [\![P]\!](\hat{\rho} t_1)(\hat{\rho} t_2)$$

$$\mathcal{M} \vDash_\rho \varphi \dot{\to} \psi := \mathcal{M} \vDash_\rho \varphi \to \mathcal{M} \vDash_\rho \psi \qquad \mathcal{M} \vDash_\rho \dot{\forall} \varphi := \forall d : D.\ \mathcal{M} \vDash_{d;\rho} \varphi$$

where $d; \rho := \lambda x.$ **if** x **is** $\mathsf{S}\, x$ **then** ρx **else** d. A formula φ is **valid** if $\forall \mathcal{M}. \forall \rho.\ \mathcal{M} \vDash_\rho \varphi$ and **satisfiable** if $\exists \mathcal{M}. \forall \rho.\ \mathcal{M} \vDash_\rho \varphi$.

Def. valid formula

Def. satisfiable formula

We call the models *naive*, because they are usually defined in a classical meta theory.

We define a parallel substitution operation $\varphi[\sigma]$ for a substitution $\sigma : \mathbb{N} \to \mathsf{tm}$ as

$$x[\sigma] := \sigma x \qquad e[\sigma] := e \qquad (f_0 t)[\sigma] := f_0(t[\sigma]) \qquad (f_1 t)[\sigma] := f_1(t[\sigma])$$

$$\dot{\bot}[\sigma] := \dot{\bot} \qquad Q[\sigma] := Q \qquad (Pt)[\sigma] := P(t[\sigma])$$

$$(\varphi \dot{\to} \psi)[\sigma] := (\varphi[\sigma]) \dot{\to} (\psi[\sigma]) \qquad (\dot{\forall}\varphi)[\sigma] := \dot{\forall}\varphi[0; \lambda x. \uparrow (\sigma x)]$$

where $\uparrow t := t[\sigma x; \mathsf{S}\, x]$.

We define minimal, intuitionistic, and classical provability using natural deduction systems.

Def. Minimal natural deduction

Minimal natural deduction $\vdash : \mathbb{L}(\mathsf{fm}) \to \mathsf{fm} \to \mathbb{P}$ is defined as follows:

$$\frac{\varphi \in \Gamma}{\Gamma \vdash \varphi} \qquad \frac{\Gamma \vdash \varphi \dot{\to} \psi \qquad \Gamma \vdash \varphi}{\Gamma \vdash \psi} \qquad \frac{\varphi :: \Gamma \vdash \psi}{\Gamma \vdash \varphi \dot{\to} \psi} \qquad \frac{\Gamma \vdash \dot{\forall}\varphi}{\Gamma \vdash \varphi[t]} \qquad \frac{\uparrow \Gamma \vdash \varphi}{\Gamma \vdash \dot{\forall}\varphi}$$

Def. intuitionistic natural deduction

Def. explosion rule

Def. classical natural deduction

Def. double negation rule

To obtain **intuitionistic natural deduction** $\vdash_i : \mathbb{L}(\mathsf{fm}) \to \mathsf{fm} \to \mathbb{P}$ we use the same rules as for $\vdash$ and add the **explosion rule** (left). To obtain **classical natural deduction** $\vdash_c : \mathbb{L}(\mathsf{fm}) \to \mathsf{fm} \to \mathbb{P}$ we use the same rules as for $\vdash$, and add the **double negation rule** (right).

$$\frac{\Gamma \vdash_i \dot{\bot}}{\Gamma \vdash_i \varphi} \qquad \frac{\Gamma \vdash_c (\varphi \dot{\to} \dot{\bot}) \dot{\to} \dot{\bot}}{\Gamma \vdash_c \varphi}$$

Def. provable formula

If $[\,] \vdash \varphi$ (or $[\,] \vdash_i \varphi$ or $[\,] \vdash_c \varphi$) we say that φ is minimally (or intuitionistically or classically) **provable**. Clearly, minimal provability implies intuitionistic provability. That intuitionistic provability implies classical provability requires weakening, which holds for all systems.

$\subseteq$ → Sec. A.1, Page 207

Fact 23.1. Let $\Gamma \subseteq \Gamma'$. Then $\Gamma \vdash \varphi \to \Gamma' \vdash \varphi$, $\Gamma \vdash_i \varphi \to \Gamma' \vdash_i \varphi$, and $\Gamma \vdash_c \varphi \to \Gamma' \vdash_c \varphi$.

Fact 23.2. If $\Gamma \vdash \varphi$, then $\Gamma \vdash_i \varphi$ and if $\Gamma \vdash_i \varphi$ then $\Gamma \vdash_c \varphi$.

[74] Forster, Kirst, and Wehr. 2020a. Completeness Theorems for First-Order Logic Analysed in Constructive Type Theory.

Furthermore, intuitionistic provability is sound w.r.t. naive Tarski models.

Fact 23.3. If $[\psi_1, \ldots, \psi_n] \vdash_i \varphi$, then $\psi_1 \dot{\to} \ldots \dot{\to} \psi_n \dot{\to} \varphi$ is valid.

In contrast, classical provability is only sound w.r.t. naive Tarski models if and only if the meta-theory is classical. A non-naive version of Tarski semantics such that classical provability is sound and for completeness the assumption of MP suffices is discussed in [74].

Fact 23.4. LEM holds if and only if $[\psi_1,\dots,\psi_n]\vdash_c \varphi$ implies that $\psi_1\dot\to\dots\dot\to\psi_n\dot\to\varphi$ is valid.

23.2 Reducing BPCP to validity and minimal and intuitionistic provability, and $\overline{\mathsf{BPCP}}$ to satisfiability

The most economical undecidability proof for validity, minimal and intuitionistic provability, and satisfiability is by doing it in one take. The proof idea is based on the observation that PCP can be expressed without inductive predicates, instead using Horn clauses as follows:

Lemma 23.5. $\mathsf{PCP}_X R$ if and only if for all $P\colon \mathbb{L}X\to\mathbb{L}X\to\mathbb{P}$ and for all $Q\colon\mathbb{P}$,

$$(\forall(u,v)\in R.\, Puv)\to(\forall(u,v)\in R.\forall xy.\, Pxy\to P(u{+\!\!+}x)(v{+\!\!+}y))\to(\forall x.\, Pxx\to Q)\to Q$$

Proof. The direction from left to right is by induction on PCP_X. The direction from right to left by instantiating $Pxy := R\triangleright xy$ and $Q := \mathsf{PCP}_X R$. ∎

We now mirror this characterisation in first-order logic and define a formula φ_R such that $\mathsf{BPCP}\,R$ implies that φ_R is minimally provable (and thus both intutionistically provable by Fact 23.2 and valid by Fact 23.3).

We then define a naive Tarski model $\mathcal{B}$ instantiating P and Q above, such that $\mathcal{B}\vDash_\rho \varphi_R$ is equivalent to BPCP_R. Thus validity of φ_R implies $\mathsf{BPCP}\,R$, and both intuitionistic and minimal provability imply $\mathsf{BPCP}\,R$ as well, again using Facts 23.2 and 23.3. Since then $\neg\mathsf{BPCP}\,R$ if and only if the negation of φ_R is satisfiable, we obtain a reduction from $\overline{\mathsf{BPCP}}$ to satisfiability.

To define φ_R we have to express ${+\!\!+}$ in terms of e, f_0, and f_1. We define an encoding $\overline{u}\colon\mathsf{tm}$ of lists $u\colon\mathbb{L}\mathbb{B}$ via a function ${+\!\!+}\colon\mathbb{L}\mathbb{B}\to\mathsf{tm}\to\mathsf{tm}$, and the formula φ_R as follows:

$$[]{+\!\!+}t := t \qquad (b::u){+\!\!+}t := \mathbf{if}\ b\ \mathbf{then}\ f_0(u{+\!\!+}t)\ \mathbf{else}\ f_1(u{+\!\!+}t) \qquad \overline{u} := u{+\!\!+}e$$

$$\Gamma_1 := [P\overline{u}\overline{v}\mid(u,v)\in R] \qquad \Gamma_2 := [\dot\forall xy.\, Pxy\dot\to P(u{+\!\!+}x)(v{+\!\!+}y)\mid(u,v)\in R]$$

$$\varphi_3 := \dot\forall x.\, Pxx\dot\to Q \qquad \varphi_R := \Gamma_1\dot\to\Gamma_2\dot\to\varphi_3\dot\to Q$$

For the direction from left to right, we show that solutions can be constructed in provability:

Fact 23.6. If $R\triangleright(u,v)$ then for all Γ, $\Gamma{+\!\!+}\Gamma_1{+\!\!+}\Gamma_2\vdash P\,\overline{u}\,\overline{v}$.

Corollary 23.7. $\mathsf{BPCP}\,R\to[]\vDash\varphi_R$

For the other direction, we define a standard model $\mathcal{B}$ such that satisfiability of φ_R in $\mathcal{B}$ allows reconstructing a solution for $\mathsf{BPCP}\,R$. The model $\mathcal{B}$ interprets e as $[]$, f as $::$, P as $R\triangleright(\_,\_)$, and Q as $\mathsf{BPCP}\,R$:

$$[\![e]\!]_{\mathcal{B}} := [] \qquad [\![f_0]\!]_{\mathcal{B}}t := \mathsf{false}::s \qquad [\![f_1]\!]_{\mathcal{B}}t := \mathsf{true}::s$$

$$[\![Q]\!]_{\mathcal{B}} := \mathsf{BPCP}R \qquad [\![P]\!]_{\mathcal{B}}t_1t_2 := R\triangleright(t_1,t_2)$$

Fact 23.8. $\hat\rho(l{+\!\!+}t) = t{+\!\!+}\hat\rho\,t$, $\hat\rho\,\overline{l} = l$, and $\mathcal{B}\vDash_\rho Pt_1t_2\leftrightarrow R\triangleright(\hat\rho\,t_1,\hat\rho\,t_2)$.

Fact 23.9. For all ρ, $\mathcal{B} \vDash_\rho \varphi_3$ and for all $\varphi \in \Gamma_1 +\!\!+ \Gamma_2$, $\mathcal{B} \vDash_\rho \varphi$.

Corollary 23.10. $\mathcal{B} \vDash_\rho \varphi_R \to \mathsf{BPCP}\, R$

Theorem 23.11. BPCP reduces to validity and minimal and intuitionistic provability:

$$\mathsf{BPCP} \preceq_m \lambda\varphi.\, \forall \mathcal{M}\rho.\, \mathcal{M} \vDash_\rho \varphi \qquad \mathsf{BPCP} \preceq_m \lambda\varphi.\, [] \vdash \varphi \qquad \mathsf{BPCP} \preceq_m \lambda\varphi.\, [] \vdash_i \varphi$$

Proof. The only interesting part is to prove that if $[] \vdash_i \phi_R$, then $\mathsf{BPCP}\, R$, which follows immediately using soundness (Fact 23.3). ∎

Note that when using a non-naive Tarski-semantics restricting to classical models, using soundness requires proving that $\mathcal{B}$ is classical, which only holds provided MP.

Fact 23.12. $\neg\mathsf{BPCP}\, R$ if and only if $\varphi_R \dot{\to} \dot{\bot}$ is satisfiable.

Corollary 23.13. $\overline{\mathsf{BPCP}}$ reduces to satisfiability: $\overline{\mathsf{BPCP}} \preceq_m \lambda\varphi.\, \exists \mathcal{M}. \forall \rho. \mathcal{M} \vDash_\rho \varphi$

23.3 Reducing BPCP to classical provability

Our proof strategy before crucially relied on soundness of provability w.r.t. naive Tarski semantics to use the model $\mathcal{B}$. Since classical provability is not constructively sound w.r.t. naive Tarski semantics, we in principle have two possibilites: First, we could devise a less naive version of Tarski semantics where every model has to validate the double negation rule, as e.g. done in [74]. However, the defined model $\mathcal{B}$ does only validate double negation under MP.

To obtain a fully constructive undecidability proof we stick to naive Tarski semantics and choose the second possible route: We define a translation φ^Q satisfying $(\vdash_c \varphi) \to (\vdash \varphi^Q)$ and $(\mathcal{B} \vDash_\rho \varphi_R^Q) \to (\mathsf{BPCP}\, R)$. The translation is a combination of the Gödel-Gentzen double negation translation [92, 89] and Friedman's A-translation [87], with A fixed to be Q.

[92] Gödel. 1930. Die Vollständigkeit der Axiome des logischen Funktionenkalküls.

[89] Gentzen. 1936. Die Widerspruchsfreiheit der reinen Zahlentheorie.

[87] Friedman. 1978. Classically and intuitionistically provably recursive functions.

$$\dot{\bot}^Q := Q \qquad Q^Q := Q \qquad (Pt_1t_2)^Q := ((Pt_1t_2) \dot{\to} Q) \dot{\to} Q$$

$$(\varphi \dot{\to} \psi)^Q := \varphi^Q \dot{\to} \psi^Q \qquad (\dot{\forall}\varphi)^Q := \dot{\forall}(\varphi^Q)$$

Lemma 23.14. $\Gamma \vdash ((\varphi \dot{\to} \bot) \dot{\to} \bot))^Q \dot{\to} \varphi^Q$

Proof. By induction on the size of φ with Γ generalised. ∎

Corollary 23.15. $\Gamma \vdash_c \varphi \to \Gamma^Q \vdash \varphi^Q$

Note that by using an alternative translation with $\dot{\bot}$ in place of Q we could reduce classical provability to intuitionistic provability. We refrain from doing so here, because this direction does not contribute anything with regards to undecidability proofs.

Theorem 23.16. BPCP reduces to classical provability: $\mathsf{BPCP} \preceq_m \lambda\varphi.\, [] \vdash_c \varphi$

Proof. We prove that $\mathsf{BPCP}\, R \leftrightarrow [] \vdash_c \varphi_R$.

The direction from left to right is immediate using Corollary 23.7 and Fact 23.2.

For the direction from right to left let $[] \vdash_c \varphi_R$. By Facts 23.15 and 23.3 we have that $\mathcal{B} \vDash_{\lambda x.\,[]} \varphi_R^Q$. Since $\varphi_R^Q = [\varphi^Q \mid \varphi \in \Gamma_1] \to [\varphi^Q \mid \varphi \in \Gamma_2] \to \varphi_3^Q \to Q$ and for all $\varphi \in \Gamma_1 +\!\!+ \Gamma_2$, $\mathcal{B} \vDash_{\lambda x.\,[]} \varphi^Q$ as well as $\mathcal{B} \vDash_{\lambda x.\,[]} \varphi_3^Q$ follows from Fact 23.9 by simple calculation, we have $\mathcal{B} \vDash_{\lambda x.\,[]} Q$ and thus $\mathsf{BPCP}\, R$. ∎

CHAPTER 24

Higher-order unification

Higher-order unification – where abstractions $\lambda x.s$ can be substituted for variables – is fundamental for many applications in modern programming languages. It is the foundation of languages such as λ-Prolog, is used in automated deduction, and is crucial for type inference in dependent type theories such as CIC for type inference. In the form in which we consider it, higher-order unification HOU is the problem of finding a substitution making two given, simply-typed Curry-style λ-terms convertible. One can also consider n-th order unification, where for free variables the nesting depth of function types in argument positions is restricted. First-order unification was proved decidable in 1965 by Robinson [201], third-order unification was proved undecidable independently by Huet [116, 117] and Lucchesi [160] by reduction from PCP, and subsequently second-order unification was proved undecidable by Goldfarb [96] by reduction from H10.

We establish undecidability based on a reduction by Dowek [60], who reduces H10 to HOU by devising characteristic typed unification equations for Church numerals. The resulting equations are of order 3 but simpler to understand and verify than Goldfarb's proof. We present a similar proof based on equations devised by Simon Spies [215], which are characteristic *untyped* unification equations.

[201] Robinson. 1965. A machine-oriented logic based on the resolution principle.

[116] Huet. 1972. Constrained Resolution: A Complete Method for Higher-Order Logic..

[117] Huet. 1973. The undecidability of unification in third order logic.

[160] Lucchesi. 1972. The undecidability of the unification problem for third order languages.

[96] Goldfarb. 1981. The undecidability of the second-order unification problem.

[60] Dowek. 2001. Higher-Order Unification and Matching..

Publications This chapter is based on

[216] Spies and Forster. "Undecidability of higher-order unification formalised in Coq" *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 2020.

24.1 Definition

We consider the syntax of the λ-calculus with Curry-style abstraction (i.e. the argument type is not annotated), application, and variables:

$$s,t : \mathsf{tm} ::= st \mid \lambda s \mid n \quad \textit{where } n : \mathbb{N}$$

As before, we use de Bruijn indices for the formal definition, but write named terms on paper.

We only consider function types and type variables, modelled by natural numbers:

$$A,B : \mathsf{ty} ::= \alpha \mid A \dot{\to} B \quad \textit{where } \alpha : \mathbb{N}$$

The simple type system $\vdash : \mathbb{L}(\mathsf{ty}) \to \mathsf{tm} \to \mathsf{ty} \to \mathbb{P}$ is defined as follows:

$$\frac{\Gamma[n] = \mathsf{Some}\, A}{\Gamma \vdash n : A} \qquad \frac{A :: \Gamma \vdash s : B}{\Gamma \vdash \lambda s : A \dot{\to} B} \qquad \frac{\Gamma \vdash s : A \dot{\to} B \qquad \Gamma \vdash t : A}{\Gamma \vdash st : B}$$

Def. renaming

Def. parallel substitution

We define **renaming** $s\langle\rho\rangle$ for $\rho : \mathbb{N} \to \mathbb{N}$ and **parallel substitution** $s[\sigma]$ for $\sigma : \mathbb{N} \to \mathsf{tm}$:

$$n\langle\rho\rangle := \rho n \qquad (\lambda s)\langle\rho\rangle := \lambda(s\langle\uparrow\rho\rangle) \qquad (st)\langle\rho\rangle := (s\langle\rho\rangle)(t\langle\rho\rangle)$$

$$n[\sigma] := \sigma n \qquad (\lambda s)[\sigma] := \lambda(s[\Uparrow\sigma]) \qquad (st)[\sigma] := (s[\sigma])(t[\sigma])$$

where $(\uparrow\rho)n :=$ **if** n **is** $\mathsf{S}\,n$ **then** $\mathsf{S}(\rho n)$ **else** 0 and $(\Uparrow\sigma)n :=$ **if** n **is** $\mathsf{S}\,n$ **then** $(\sigma n)\langle\lambda x.\,\mathsf{S}\,x\rangle$ **else** 0.

We write $\Delta \vdash \sigma : \Gamma$ for $\forall nA.\ \Gamma[n] = \mathsf{Some}A \to \Delta \vDash \sigma n : A$.

In Coq, we generate the definition of substitution and its basic properties using the Autosubst 2 tool [218]. As a consequence, basic results depend on the functional extensionality axiom Fext. The dependency could be eliminated by improvements to the tool or manual adaption of the generated code.

[218] Stark, Schäfer, and Kaiser. 2019. Autosubst 2: reasoning with multi-sorted de Bruijn terms and vector substitutions.

Fext → Sec. 7.3, Page 70

We define β-equivalence $\equiv : \mathsf{tm} \to \mathsf{tm} \to \mathbb{P}$ as the reflexive, transitive, symmetric closure of β-reduction $\succ : \mathsf{tm} \to \mathsf{tm} \to \mathbb{P}$:

$$\frac{t_1 \succ t_1'}{t_1 t_2 \succ t_1' t_2} \qquad \frac{t_2 \succ t_2'}{t_1 t_2 \succ t_1 t_2'} \qquad \frac{}{(\lambda t_1) t_2 \succ t_1[\lambda x.\ \textbf{if } x \textbf{ is } \mathsf{S}\,x \textbf{ then } x \textbf{ else } t_2]}$$

$$\frac{}{t \equiv t} \qquad \frac{t_1 \succ t_1' \qquad t_1' \equiv t_2}{t_1 \equiv t_2} \qquad \frac{t_2 \equiv t_1}{t_1 \equiv t_2}$$

It is well-known that the simply-typed λ-calculus is strongly normalising. We only need weak normalisation:

Fact 24.1. If $\Gamma \vdash t : A$ then there exists v such that $t \equiv v$, $\Gamma \vdash v : A$, and $\neg\exists t'.\ v \succ t'$.

If $t_1[\sigma] \equiv t_2[\sigma]$ we say that σ **unifies** (t_1, t_2).

The decision problem higher-order unification HOU asks whether two terms of the same types can be unified by substitutions:

$$\mathsf{HOU}\ (\Gamma{:}\,\mathbb{L}(\mathsf{ty}), A{:}\,\mathsf{ty}, t_1{:}\,\mathsf{tm}, t_2{:}\,\mathsf{tm}, H_1 : (\Gamma \vdash t_1{:}A), H_2 : (\Gamma \vdash t_2{:}A)) :=$$

$$\exists\Delta{:}\,\mathbb{L}(\mathsf{ty}).\exists\sigma{:}\,\mathbb{N} \to \mathsf{tm}.\ \Delta \vdash \sigma : \Gamma \wedge t_1[\sigma] \equiv t_2[\sigma]$$

Similarly, system unification SHOU asks whether all pairs in a list can be pointwise unified:

$$\mathsf{SHOU}\,(\Gamma{:}\,\mathbb{L}(\mathsf{ty}), L{:}\,\mathbb{L}(\mathsf{tm} \times \mathsf{tm} \times \mathsf{ty}), H : (\forall(t_1, t_2, A) \in L.\ \Gamma \vdash t_1 : A \wedge \Gamma \vdash t_2 : A)) :=$$

$$\exists\Delta{:}\,\mathbb{L}(\mathsf{ty}).\exists\sigma{:}\,\mathbb{N} \to \mathsf{tm}.\ \Delta \vdash \sigma : \Gamma \wedge \forall(t_1, t_2, A) \in L.\ t_1[\sigma] \equiv t_2[\sigma]$$

24.2 Reducing SHOU to HOU

To reduce SHOU to HOU we have to encode a list of terms as single term. We define

$$f[(s_1, t_1, A_1), \ldots, (s_n, t_n, A_n)] := (\lambda x.\ x\, s_1 \ldots s_n, \lambda x.\ x\, t_1 \ldots t_n)$$

$$g[(s_1, t_1, A_1), \ldots, (s_n, t_n, A_n)] := A_1 \dot{\to} \ldots \dot{\to} A_n \dot{\to} \alpha$$

where $\alpha := 0$.

We fix $\Gamma{:}\,\mathbb{L}(\mathsf{ty})$, $L{:}\,\mathbb{L}(\mathsf{tm} \times \mathsf{tm} \times \mathsf{ty})$ such that $\forall(t_1, t_2, A) \in L.\ \Gamma \vdash t_1 : A \wedge \Gamma \vdash t_2 : A$.

Fact 24.2. $\Gamma \vdash f\,L : g\,L \to \alpha$

Fact 24.3. If $\sigma : \mathbb{N} \to \mathsf{ty}$ such that $\Delta \vdash \sigma : \Gamma$ we have $\forall (t_1, t_2, A) \in L.\ t_1[\sigma] \equiv t_2[\sigma]$ if and only if $(f\ t_1)[\sigma] \equiv (f\ t_2)[\sigma]$.

Theorem 24.4. $\mathsf{SHOU} \preceq_m^{\mathsf{Fext}} \mathsf{HOU}$ and thus $\mathsf{Fext} \to \mathcal{U}(\mathsf{HOU})$.

$\preceq_m^{\mathsf{Fext}}$ → Sec. 19.3, Page 145

24.3 Reducing H10C to SHOU

We reduce diophantine constraints H10C to a system of equations by using Church numerals, where

$$[\![n]\!] := \lambda af.\ f^n a \qquad f^0 a := f \qquad f^{\mathsf{S}\,n} a := f(f^n a)$$

Fact 24.5. If $[\![n_1]\!] \equiv [\![n_2]\!]$ then $n_1 = n_2$.

Church numerals can be assigned a simple type.

Fact 24.6. For all A, $\Gamma \vdash [\![n]\!] : A \dot{\to} (A \dot{\to} A) \dot{\to} A$

We use standard definitions of addition and multiplication [8], which Hinze [111] attributes to Rosser:

$$\mathsf{add}\, t_1 t_2 := \lambda af.\ t_1 (t_2 a f) f \qquad \mathsf{mul}\, t_1 t_2 := \lambda af.\ t_1 a (\lambda b.\ t_2 b f)$$

[8] Barendregt, Dekkers, and Statman. 2013. Lambda calculus with types.

[111] Hinze. 2005. Church numerals, twice! (theoretical pearl).

Fact 24.7. $\mathsf{add}[\![n_1]\!][\![n_2]\!] \equiv [\![n_1 + n_2]\!]$ and $\mathsf{mul}[\![n_1]\!][\![n_2]\!] \equiv [\![n_1 \cdot n_2]\!]$.

The inverse of $[\![\cdot]\!]$ is computable.

Fact 24.8. There is $\mathsf{inv} : \mathsf{tm} \to \mathbb{ON}$ such that **if** $\mathsf{inv}\, t$ **is** $\mathsf{S}\, n$ **then** $t = [\![n]\!]$ **else** $\neg \exists n.\ s = [\![n]\!]$.

Dowek [60] observes that $t : \alpha \dot{\to} (\alpha \dot{\to} \alpha) \dot{\to} \alpha$ is a Church numeral if and only if

$$\lambda z.tz(\lambda y.y) \equiv \lambda z.z.$$

We instead use *Spies equations*, which also work in an untyped setting. Recall that a Church numeral $[\![n]\!]$ expects two arguments f and a and then applies f to a for n times. Thus, the equations characterise that t is a Church numeral by observing the commutative property of iteration, i.e. $\forall f\, a.\ f([\![n]\!] a f) \equiv [\![n]\!](f\, a) f$.

Fact 24.9. Let t be normal. We have that $t = [\![n]\!]$ for some n if and only if $(\lambda af.\ f(taf)) \equiv (\lambda af.\ t(f\, a) f)$.

We can then define the reduction function where again $\dot{\alpha} := 0$:

$$\dot{\mathbb{N}} := \alpha \dot{\to} (\alpha \dot{\to} \alpha) \dot{\to} \alpha \qquad Nx := (\lambda af.\ f(taf), \lambda af.\ t(f\, a) f, \dot{\mathbb{N}})$$

$$f[\,] := [\,] \qquad f(x \dot{+} y \doteq z :: C) := [Nx, Ny, Nz, (\mathsf{add}\, x\, y, z, \dot{\mathbb{N}})] +\!\!+ f\, C$$

$$f(x \doteq 1 :: C) := [Nx, (x, [\![1]\!], \dot{\mathbb{N}})] +\!\!+ f\, C \qquad f(x \times y \doteq z :: C) := [Nx, Ny, Nz, (\mathsf{mul}\, x\, y, z, \dot{\mathbb{N}})] +\!\!+ f\, C$$

Theorem 24.10. $\mathsf{H10C} \preceq_m^{\mathsf{Fext}} \mathsf{SHOU}$ and thus $\mathsf{Fext} \to \mathcal{U}(\mathsf{SHOU})$.

Proof. Let a list of constraints C be given and let $\mathsf{vars}\, l$ be the list of variables occurring in l. We prove $\mathsf{H10C}\, C \leftrightarrow \mathsf{SHOU}([\dot{\mathbb{N}} \mid x \in \mathsf{vars}\, C], f\, C, H)$ where H proves that

$$\forall (t_1, t_2, A) \in f\, C.\quad ([\dot{\mathbb{N}} \mid x \in \mathsf{vars}\, C] \vdash t_1 : A) \ \wedge\ ([\dot{\mathbb{N}} \mid x \in \mathsf{vars}\, C] \vdash t_1 : A).$$

For the direction from left to right let φ s.t. $\varphi \vDash C$ be given. Let $\Delta := []$ and $\sigma x := [\![\varphi x]\!]$. The claim follows by Facts 24.6 and 24.7 and induction on C.

For the converse direction, let σ unify fC and $\Delta \vdash \sigma : [\dot{\mathbb{N}} \mid x \in \mathsf{vars}\, C]$. By the latter and Fact 24.1 let L be the list of normal forms of $[\sigma x \mid x \in \mathsf{vars}\, C]$. Then $\varphi n :=$ **if** $L[n]$ **is** Some t **then if** inv t **is** Some n **then** n **else** 0 **else** 0 solves C by Facts 24.8, 24.9, and 24.7. ∎

CHAPTER 25

The Coq Library of Undecidability Proofs

Since 2018, the problems described in this part of the thesis have formed the foundation of the *Coq Library of Undecidability Proofs*, a collaborative effort of several researchers at various institutions spanning more than 110.000 lines of code.

The library contains various undecidable problems, which can roughly be classified into *seed problems*, *advanced problems*, and *target problems*. Seed problems have simple definitions and work well as starting point for reductions, i.e. for a seed S and a problem p, it is often easy to establish e.g. $S \preceq_m p$. Typical seed problems include PCP and H10C. Target problems are expressive problems, usually with more involved definitions, and work well as end points for reductions, i.e. for a target problem T and a problem p, it is often easy to establish e.g. $p \preceq_m T$. Thus, once a target problem is proved undecidable by many-one reduction from the halting problem, it can be used for many-one equivalence proofs. Typical target problems include first-order logic or L. Advanced problems are problems where a machine-checked undecidability proof is of interest independent to its use for other proofs. Typical advanced problems part of this thesis include higher-order unification, but more generally are problems like semi-unification or type-checking and typability for System F.

We first give a brief statistical overview over the library. We then survey work which is part of the undecidability library without the involvement of the author (Section 25.2), give a comparison of the Undecidability Library with other libraries in the Coq ecosystem (Section 25.3), and directions for future work (Section 25.4).

25.1 Statistics

At the time of writing, the Coq Library of Undecidability proofs spans 117k lines of code (LoC), with 51k lines of specification and 66k lines of proofs, i.e. with a ratio of 44% specification vs. 56% proofs. Figure 25.2 shows how the code is organised into sub-projects.

The library was founded by the author of this thesis and Dominique Larchey-Wendling in October 2018, but plans to do so date back to early 2018 [80]. Initially, the library comprised the code accompanying four papers [83, 71, 81, 73]. The library is collaboratively developed via GitHub, were Dominique Larchey-Wendling and the author of this thesis act as maintainers. Collaboration is arranged via pull requests, of which 109 where filed since 2018, and discussed in issues (19 in total as of May 2021). Figure 25.1 shows an overview of contributed lines per author.

Continuous integration (CI) first using Travis and then GitHub's built-in CI tremendously speeds up the checking of PRs by maintainers. CI has checked over 800 intermediate states of the repository, with successful runs taking around 25 minutes.

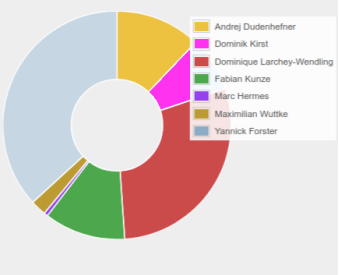


Figure 25.1.: Contribution statistics

TM:	23364	MuRec:	4529	CounterMachines:	839
TRAKHTENBROT:	10635	MinskyMachines:	3883	HilbertCalculi:	769
L:	10202	ILL:	2623	SemiUnification:	652
HOU:	9755	FOLP:	2234	SeparationLogic:	575
FOL:	8142	StringRewriting:	1704	DiophantineConstraints:	507
H10:	7300	PCP:	1127	PolynomialConstraints:	227
StackMachines:	5820	FRACTRAN:	1083	Shared:	15001
SystemF:	4922	SetConstraints:	846	Synthetic:	973

Figure 25.2.: Line counts of subdirectories

25.2 Other machine-checked undecidability proofs

Several results in the library were contributed independently, see fig. 25.3.

Logic Kirst and Larchey-Wendling prove Trakthenbrot's theorem [227] in Coq, i.e. the undecidability of satisfiability for first-order logic w.r.t. a model with finite domain FSAT by reduction from BPCP [129]. In an extended version, they also verify a reduction from FSAT to satisfiability in minimal separation logic MSLSAT and separation logic SLSAT [130].

Forster and Larchey-Wendling prove the undecidability of entailment in elementary intuitionistic linear logic EILL and intuitionistic linear logic ILL [81]. The undecidability of ILL was first proved by Lincoln, Mitchell, Scedrov, and Shankar [157] by reduction from and-branching two-counter machines without zero-test. The undecidability of EILL was first proved by Larchey-Wendling and Galmiche [154] by reduction from two-counter machines. Larchey-Wendling mechanises a reduction from ILL to entailment in classical linear logic CLL. The undecidability of entailment in (intuitionistic) multiplicative sub-exponential linear logic IMSELL by reduction from $\mathsf{Halt}_{\mathsf{FRACTRAN}}$ was proved and mechanised by Larchey-Wendling [151]. The undecidability of IMSELL before was proved on paper via reduction from the halting problem of two register counter machines [28].

Kirst and Hermes [128] mechanise the undecidability of naive Tarski validity for Peano arithmetic PA, provability in Heyting arithmetic HA, naive Tarski validity in ZF set theory, and provability in a system similar to IZF (the arithmetical problems are displayed as Arith in the diagram). They conclude via Post's proof idea [192] that if HA or respectively IZF are negation-complete, $\mathsf{Halt}_{\mathsf{TM}}$ would be synthetically decidable, i.e. HA and IZF are synthetically negation-incomplete.

[227] Trakhtenbrot. 1950. The impossibility of an algorithm for the decidability problem on finite classes.

[129] Kirst and Larchey-Wendling. 2020. Trakhtenbrot's Theorem in Coq.

[130] Kirst and Larchey-Wendling. 2021. Trakhtenbrot's Theorem in Coq: Finite Model Theory through the Constructive Lens.

[81] Forster and Larchey-Wendling. 2019. Certified undecidability of intuitionistic linear logic via binary stack machines and Minsky machines.

[157] Lincoln et al.. 1992. Decision problems for propositional linear logic.

[154] Larchey-Wendling and Galmiche. 2010. The undecidability of boolean BI through phase semantics.

[151] Larchey-Wendling. 2021. Synthetic Undecidability of MSELL via FRACTRAN mechanised in Coq.

[28] Chaudhuri. 2018. Expressing additives using multiplicatives and subexponentials.

[128] Kirst and Hermes. 2021b. Synthetic Undecidability and Incompleteness of First-Order Axiom Systems in Coq.

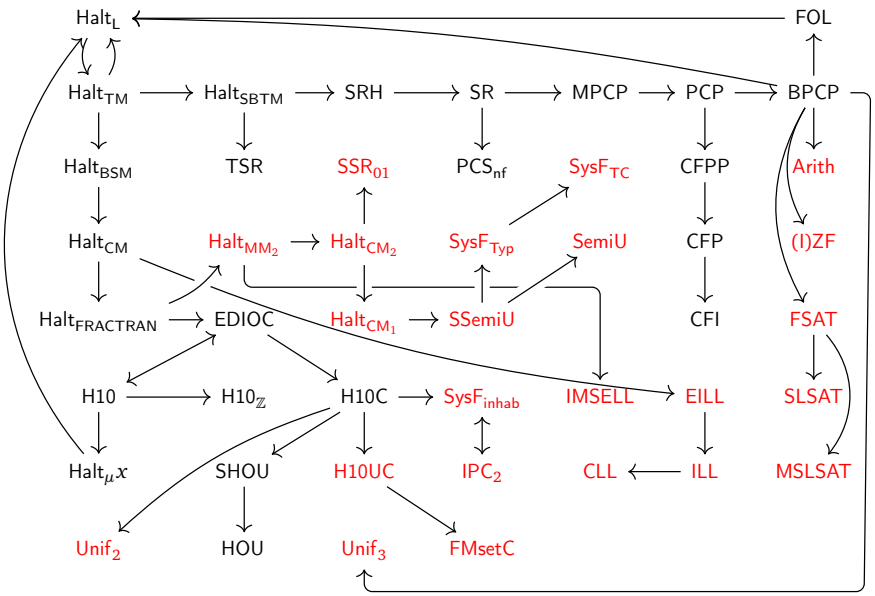


Figure 25.3.: overview of the results currently in the library. In red: results without the involvement of the author of this thesis.

Semi-Unification Semi-unification was proved undecidable by Kfoury, Tiuryn, and Urzyczyn [126] via reduction from the immortality problem of Turing machines. Dudenhefner verifies a many-one reduction from unboundedness of a certain form of counter machines to simple semi unification (SSemiU) and semi unification (SemiU) [61]. He sketches a weak truth-table reduction from the immortality problem of Turing machines to unboundedness, relying on the fan theorem. In follow-up work, Dudenhefner then verifies a constructive many-one reduction from the halting problem of two-counter machines ($\mathsf{Halt}_{\mathsf{MM}_2}$), via a more specialised problem of two-counter machines ($\mathsf{Halt}_{\mathsf{CM}_2}$) and via a halting problem for specialised one-counter machines ($\mathsf{Halt}_{\mathsf{CM}_1}$) to unboundedness [62].

[126] Kfoury et al.. 1993. The undecidability of the semi-unification problem.

[61] Dudenhefner. 2020. Undecidability of Semi-Unification on a Napkin.

[62] Dudenhefner. 2021a. Constructive Many-one Reduction from the Halting Problem to Semi-unification.

Typed λ-calculi Spies and Forster prove the undecidability of 2nd order (Unif_2) and 3rd order unification (Unif_3) by reduction from H10C and BPCP, respectively [216].

Dudenhefner and Rehof verify a reduction from H10C to the inhabitation problem for System F $\mathsf{SysF}_{\mathsf{inhab}}$ [64]. Equivalently, the problem can be seen as provability in the implicational fragment of intuitionistic second-order propositional logic IPC_2, originally proved undecidable by reduction from provability in first-order logic by Löb [158]. Dudenhefner verifies a many-one reduction from simple semi unification to the typability problem of System F $\mathsf{SysF}_{\mathsf{Typ}}$, and further to the typechecking problem for System F $\mathsf{SysF}_{\mathsf{TC}}$ [63]. The proof is a less versatile simplification of the original undecidability proof on paper by Wells [234].

[216] Spies and Forster. 2020. Undecidability of higher-order unification formalised in Coq.

[64] Dudenhefner and Rehof. 2018. A Simpler Undecidability Proof for System F Inhabitation.

[158] Löb. 1976. Embedding first order predicate logic in fragments of intuitionistic logic.

[63] Dudenhefner. 2021b. The Undecidability of System F Typability and Type Checking for Reductionists.

[234] Wells. 1999. Typability and type checking in System F are equivalent and undecidable.

Projects outside of the library We are aware of two projects outside of the context of the library, which are concerned with the verification of reductions.

The first is a reduction of the bounded quantification problem in $F^{-}_{<:}$ (a version of System F with subtyping) to subtyping in $D_{<:}$ by Hu and Lhoták [115], a subset of the core calculus DOT of the Scala programming language. The proofs are mechanised in Agda. Porting the proofs to Coq seems feasible, but would first need a result for subtyping in System F.

The second is a simulation of μ-recursive functions in a choreographic language [42], mechanised in Coq. An integration into the library seems feasible.

[115] Hu and Lhoták. 2019. Undecidability of $D_{<:}$ and its decidable fragments.

[42] Cruz-Filipe, Montesi, and Peressotti. 2021. Formalising a Turing-Complete Choreographic Language in Coq.

25.3 Other Coq libraries

Lean `mathlib`	544 k	GeoCoq	134 k
UniMath	182 k	Coq Library of Undecidability Proofs	117k
Gaia	154 k	`mathcomp`	96k
CoRN	151 k	Iris (core repository)	43k

Table 25.4.: Lines of Code of selected Coq projects.[1]

The `awesome-coq` repository [34] curates a list of Coq libraries, plugins, tools, and resources. As of May 7th 2020, it lists 18 projects in the category "Libraries", amongst them the *Coq Library of Undecidability Proofs*. Of all the libraries listed, the *Coq Library of Undecidability Proofs* is the biggest in terms of lines of code. This is however also due to the fact that the ecosystems around `mathcomp` [162] or Iris [141] are developed in separate repositories.

The category "Type Theory and Mathematics" lists 19 projects. Here, the Undecidability Library is surpassed by the *UniMath* project (covering univalent mathematics) [232], the *GeoCoq* project (covering "a formalization of geometry in Coq based on Tarski's axiom system") [221], the *Gaia* project (covering results from "Elements of Mathematics by N. Bourbaki in Coq using the Mathematical Components library, including set theory and number theory") [100], and the *CoRN* project (the "Coq Repository at Nijmegen") [53, 41].

[232] Voevodsky, Ahrens, Grayson, et al.. [n. d.]. UniMath — a computer-checked library of univalent mathematics.

[41] Cruz-Filipe, Geuvers, and Wiedijk. 2004. C-CoRN, the constructive Coq repository at Nijmegen.

The *mathlib* is the centralised mathematical library of the Lean proof assistant. As of December 2019, it contained 174k LoC [167]. As of May 7th 2020, it contains 544k LoC.

25.4 Future work

Pierce proves undecidability of subtyping in System F, called $F_{<:}$ [185], by reduction from the halting problem of counter machines. A machine-checked proof of the reduction seems feasible, especially since it could potentially be simplified by basing it on Dudenhefner's $\mathsf{Halt}_{\mathsf{CM}_2}$ problem [62] or Larchey-Wendling's $\mathsf{Halt}_{\mathsf{MM}_2}$ problem.

[185] Pierce. 1994. Bounded quantification is undecidable.

[62] Dudenhefner. 2021a. Constructive Many-one Reduction from the Halting Problem to Semi-unification.

A seed problem not yet included in the library is Wang tiling, proved undecidable by reduction from $\overline{\mathsf{Halt}_{\mathsf{TM}^1}}$ by Berger [19]. Wang tilings are frequently used in undecidability proofs for logics, for instance for the Entscheidungsproblem [24]. A machine-checked proof of the reduction seems feasible in principle, but it is an interesting open question whether another seed problem simplifies the reduction. Since the reduction seems to make crucial use of determinism of Turing machines, problems like PCP seem to be no obvious alternative, but counter machine problems like $\mathsf{Halt}_{\mathsf{CM}_2}$ or $\mathsf{Halt}_{\mathsf{MM}_2}$ might work better.

[19] Berger. 1966. The undecidability of the domino problem.

[24] Börger, Grädel, and Gurevich. 2001. The classical decision problem.

The undecidability proof of $\lambda\Pi$ typability due to Dowek [59] starts at third-order unification Unif_3 and might be a candidate for an elegant machine-checked proof.

[59] Dowek. 1993. The undecidability of typability in the lambda-pi-calculus.

Catt and Norrish [27] verify the undecidability of the problem N of nonrandomness of numbers defined via Kolmogorov complexity [138, 139], based on the λ-calculus in HOL4. The problem N does not many-one reduce from the halting problem, since it is a simple predicate [180, III.2.12]. However, Kummer [146] proves that N is truth-table complete. Thus, a synthetic undecidability proof of N defined via Kolmogorov complexity of e.g. L, by truth-table reduction from the halting problem of L might be feasible, but likely requires machinery like fixed-point theorems for L, which however are already machine-checked [83]. A fully synthetic direct proof of $\neg\mathcal{D}N$ using SCT might also be feasible and easier.

[27] Catt and Norrish. 2021. On the formalisation of Kolmogorov complexity.

[138] Kolmogorov. 1963. On tables of random numbers.

[139] Kolmogorov. 1965. Three approaches to the quantitative definition of information.

[180] Odifreddi. 1992. Classical recursion theory: The theory of functions and sets of natural numbers.

[146] Kummer. 1996. On the complexity of random strings.

[1] All lines of code counted with the `cloc` tool [45].

Part IV

Programming in the call-by-value λ-calculus

CHAPTER 26

Introduction: Programming in the call-by-value λ-calculus

In Parts I and III we have obtained result in computability theory completely independent of models of computation, whereas Part II was agnostic towards a model and covered various different models. In particular, in Part II we proved that computability and consequently *un*computability results can be transported between models.

However, there are results for which working in a concrete model of computation cannot be avoided. In this chapter, we will establish a universal machine, prove that there exists a predicate of type $\mathbb{N}\to\mathbb{P}$ which is not decidable in any of the covered models of computation, and prove the S_n^m theorem. For all of these results, it helps to focus on one fixed model and develop powerful tools for working in the model as easy as possible. Furthermore, singling out one model eases equivalence proofs, where one wants to show that a model M is Turing-equivalent to the models discussed or that a problem p is m-equivalent to the problems discussed.

We identify the weak call-by-value λ-calculus L as a sweet spot for concrete results in computability (and even complexity) theory. In particular, this is due to the fact that programming in L is significantly easier than in any of the other models covered in this thesis, and seems superior also in comparison to other λ-calculi. Several aspects of L contribute to this fact.

First, due to its weak call-by-value semantics, programming in L is similar to programming in general-purpose programming languages. By using our certifying extraction mechanism, which fully automatically extracts simply-typed Coq functions into λ-terms together with a proof of correctness, manual implementation and verification of terms become rare. But since the tool only supports total functions, for partial functions manual verification is occasionally necessary and greatly eased by an expressive language with intuitive semantics.

Secondly, we define L using simple, capturing, point-wise de Bruijn substitution. Parallel substitution as in the full λ-calculus is more complicated to define, since it is not structurally recursive. Thus, implementing and verifying an abstract machine as in Chapter 11 for the simulation of L on Turing machines is considerably eased by a substitution operation which is easy to implement.

Thirdly, L is a reasonable model of computation for time and space complexity theory [77]. In fact, the simulation of L on Turing machines we covered in Chapter 12 only has polynomial overhead in the number of β-steps the L-term needs to find a normal-form [78]. We do not cover this time complexity property, but note that the work by Gäher and Kunze on a machine-checked proof of the Cook-Levin theorem [88] relies on the property.

In this part of the thesis, we will largely use L for equivalence proofs. With the results we already have, proving that a model of computation M is Turing-equivalent amounts to proving that computability in any of the models covered in Part II implies computability in M, and vice versa that computability in M implies computability in L.

[77] Forster, Kunze, and Roth. 2020b. The weak call-by-value λ-calculus is reasonable for both time and space.

[78] Forster, Kunze, Smolka, and Wuttke. 2021c. A Mechanised Proof of the Time Invariance Thesis for the Weak Call-by-value λ-Calculus.

[88] Gäher and Kunze. 2021. Mechanising Complexity Theory: The Cook-Levin Theorem in Coq.

Similarly, a textbook proof that the problems discussed in Part III are many-one equivalent would usually show that they are intuitively enumerable, which implies that they reduce to Turing machine halting via the Church-Turing thesis. Uses of the informal Church-Turing thesis or the formal axiom CT can be circumvented in our setting by using the certifying extraction framework, which allows extracting synthetic enumerators to L.

Lastly, we analyse notions of computability and CT defined using L. By verifying the L-computability of the many-one reductions established in Part II, it would in principle be possible to prove that none of the problems is decidable in any of the models covered in Part II.

26.1 Outline

We give a high-level overview of a certifying extraction mechanism in Chapter 27. The extraction allows one to turn a simply-typed, non-dependent Coq function f into a λ-term t_f, and also generates a certificate that the term indeed computes f. We use the certifying extraction framework for the following results:

- We obtain a verified universal machine for L by extracting a step-indexed interpreter and implementing an unbounded minimisation operation,
- we show that L can simulate Turing machines by giving a proof that every function with finite and discrete domain and discrete co-domain is L-computable,
- we show that L can simulate μ-recursive functions by extracting a step-indexed interpreter for the untyped, syntactic shape of μ-recursive functions and conclude that formulating CT for L, Turing machines, binary stack machines, counter machines, FRACTRAN, Diophantine equations, or μ-recursive functions is equivalent,
- we show that every L-enumerable problem reduces to $\mathsf{Halt}_{\mathsf{L}}$, and that thus provability in minimal first-order logic is many-one equivalent to $\mathsf{Halt}_{\mathsf{L}}$, and
- we show that every L-semi-decidable problem reduces to $\mathsf{Halt}_{\mathsf{L}}$, and that thus the boolean Post's correspondence problem BPCP is many-one equivalent to $\mathsf{Halt}_{\mathsf{L}}$.

Lastly, in Chapter 29 we define CT using L and give a machine-checked proof of the S_n^m theorem for L, yielding a proof that CT defined using L implies SCT as introduced in Section 6.2, and briefly discuss directions for a future mechanised consistency proof of CT in CIC.

26.2 Related work

For related work regarding simulation proofs of models of computation, see Section 10.2. In Section 27.6 we give related work regarding extraction and certifying compilation.

26.3 Mechanisation in Coq

The Coq code for this chapter is partially contributed to the Coq Library of Undecidability proofs, partially part of the artifacts of papers, and can also be found on the following website:

`https://ps.uni-saarland.de/~forster/thesis`

The central theorems in this part of the pdf of this thesis are hyperlinked with the html-version of the Coq code, indicated by a clickable ✣-symbol.

CHAPTER 27

Certifying Extraction

When working in any model of computation, one is frequently in the situation that one wants to implement a function f as a program in the model. Textbook proofs usually gloss over this part of proofs, of which Post assessed that once the informal proof not mentioning concrete programs was "gotten, transforming it into the formal proof [mentioning programs] turned out to be a routine chore" [189]. In a proof assistant, leaving out these proofs is no option. We discussed one possible way out in Part I by the axiom CT, stating that a program can be found for every function $f : \mathbb{N} \to \mathbb{N}$. We here discuss a second route, namely a certifying extraction framework which allows obtaining programs in the weak call-by-value λ-calculus together with a correctness proof corresponding to almost arbitrary functions of CIC. By using the framework, proofs do not depend on axioms and internally λ-terms are constructed and verified, but this tedium is largely hidden from the user and carried out automatically.

[189] Post. 1944. Recursively enumerable sets of positive integers and their decision problems.

Turning functions of the CIC into L-terms is a form of *extraction*, since CIC is a rich λ-calculus itself (with inductive types, a native pattern matching construct, and a fixed-point construct). The certifying extraction framework applies to a simply-typed, non-dependent subset of CIC without mutual or nested inductive types. Our framework is implemented as a plugin in the Coq proof assistant, relying on tools of the MetaCoq project [211] and elaborated Ltac programming to obtain automated verification tactics. The tactics were implemented by Fabian Kunze and are not covered here.

[211] Sozeau, Anand, Boulier, Cohen, Forster, Kunze, Malecha, Tabareau, and Winterhalter. 2020. The MetaCoq Project.

In this chapter, we introduce equational reasoning for L, Scott encodings of arbitrary first-order datatypes in L, and an extraction relation $f \rightsquigarrow t_f$ stating that t_f behaves like an extraction of f, i.e. that t_f computes f. To explain how to work with the framework, we verify a self-interpreter for L and introduce notions like decidability, semi-decidability, and enumerability in terms of L.

Publications The chapter is based on

[76] Forster and Kunze. "A certifying extraction with time bounds from Coq to call-by-value λ-calculus." *International Conference on Interactive Theorem Proving*. 2019.

[84] Forster and Smolka. "Call-by-value lambda calculus as a model of computation in Coq." *Journal of Automated Reasoning* 63.2 (2019): 393-413.

27.1 Equational reasoning

We have presented L in a big-step evaluation semantics. For equational reasoning one needs an equivalence relation $\equiv : \mathsf{tm}_\mathsf{L} \to \mathsf{tm}_\mathsf{L} \to \mathbb{P}$. We define $\equiv$ in terms of a small-step semantics for L:

tm_L → Sec. 11.1, Page 109

$$\frac{}{(\lambda s)(\lambda t) \succ s^0_{\lambda t}} \qquad \frac{s \succ s'}{st \succ s't} \qquad \frac{t \succ t'}{st \succ st'}$$

s^n_u → Sec. 11.1, Page 109

Here, s^n_u is the capturing de Bruijn substitution operation.

We then define $\equiv$ as the reflexive, transitive, symmetric closure of $\succ$:

$$\frac{}{s \equiv s} \qquad \frac{s \succ t}{s \equiv t} \qquad \frac{s \equiv u \qquad u \equiv t}{s \equiv t} \qquad \frac{s \equiv t}{t \equiv s}$$

▷ → Sec. 11.1, Page 109

Fact 27.1. If s is closed, then $s \triangleright t \leftrightarrow s \equiv t \wedge \exists u.\ t = \lambda u$.

Corollary 27.2. If $s \triangleright t$ and $s \equiv s'$, then $s' \triangleright t$.

To perform recursions, we introduce a recursion function

$$\rho s := \lambda a.\ (\lambda xy.\ y(\lambda z.\ xxyz)\,(\lambda xy.\ y(\lambda z.xxyz))\,sa).$$

Theorem 27.3. If s and t are closed abstractions, ρs is a closed abstraction and $\rho s t \equiv s(\rho s)t$.

27.2 Scott encodings

[204] Scott. 1968. A system of functional abstraction.

To encode natural numbers, booleans, and other first-order types as L-terms, we rely on Scott encodings [204]. The idea behind Scott encodings is that case analysis is by application.

For instance, if $\overline{b}$ encodes a boolean $b\colon\mathbb{B}$, and we want to perform a case analysis on b which proceeds with a term s_1 if $b = \mathsf{true}$ and a term s_2 otherwise, then $\overline{b}\ s_1\ s_2$ implements this case analysis.

Similarly, if $\overline{n}$ encodes a natural number $n\colon\mathbb{N}$, and we want to perform a case analysis on n which proceeds with a term $s_2\overline{m}$ if $n = \mathsf{S}\,m$ and s_1 otherwise, then $\overline{n}\ s_1\ s_2$ implements this case analysis.

We write ε_X for the Scott encoding function on an encodable type X. The Scott encoding functions for booleans, natural numbers, and for L-terms are defined as follows:

$$\varepsilon_{\mathbb{B}}\mathsf{true} := \lambda x_1 x_2.\ x_1 \qquad \varepsilon_{\mathbb{B}}\mathsf{false} := \lambda x_1 x_2.\ x_2$$

$$\varepsilon_{\mathbb{N}}0 := \lambda x_1 x_2.\ x_1 \qquad \varepsilon_{\mathbb{N}}(\mathsf{S}\,n) := \lambda x_1 x_2.\ x_2(\varepsilon_{\mathbb{N}}n)$$

$$\varepsilon_{\mathsf{tm_L}}n := \lambda x_1 x_2 x_2.\ x_1(\varepsilon_{\mathbb{N}}n) \qquad \varepsilon_{\mathsf{tm_L}}(st) := \lambda x_1 x_2 x_3.\ x_2(\varepsilon_{\mathsf{tm_L}}s)(\varepsilon_{\mathsf{tm_L}}t)$$

$$\varepsilon_{\mathsf{tm_L}}(\lambda s) := \lambda x_1 x_2 x_3.\ x_3(\varepsilon_{\mathsf{tm_L}}s)$$

[171] Mogensen. 1994. Efficient Self-Interpretation in Lambda Calculus.

Scott encoding for terms was first used by Mogensen [171], but using a HOAS approach in the Scott encoding rather than relying on de Bruijn indices.

One can define Scott encodings for all first-order types, where an inductive type is first-order if all its parameters, indices, and all argument types of all constructors are first-order [184]. In general, for a first-order inductive type X with n constructors, the constructor c_i of index i which takes a arguments of types $X_1,\dots,X_a$ has Scott encoding $\varepsilon_X(c_i x_1 \dots x_a) := \lambda y_1 \dots y_n.y_i(\varepsilon_{X_1}x_1)\dots(\varepsilon_{X_a}x_a)$.

Fact 27.4. For an inductive type X with n constructors with a constructor c_i of index i which takes a arguments, and for closed abstractions $t_1,\dots,t_a$ and $s_1,\dots,s_n$ we have that

$$\varepsilon_X(c_i t_1 \dots t_a) \text{ is closed and } \varepsilon_X(c_i t_1 \dots t_a)\,s_1 \dots s_n \equiv s_i\ t_1 \dots t_a.$$

We use the meta-programming tools of MetaCoq to implement Coq commands which on input X generate encoding functions ε_X and proofs of the equations as described in Fact 27.4. As a result, using functions on datatypes other than natural numbers is almost automatic, which significantly eases programming.

27.3 Extraction

We now introduce a meta-logical relation $x \rightsquigarrow_X t_x$ for a CIC term $x : X$ and an L-term t_x. If $x \rightsquigarrow_X t_x$, we say that x *extracts to* t_x, or vice versa that t_x *computes* x. If we do not want to make t_x explicit, we say that x is L-*computable*.

By "meta-logical" we want to hint that the relation is a logical relation in the usual sense, but also that is defined only on the meta-level, not in CIC.

$$\frac{X \text{ is first-order}}{x \rightsquigarrow_X \varepsilon_X x}$$

$$\frac{t_f \text{ is a closed abstraction} \qquad \forall x : X.\ \forall t_x : \mathsf{tm}_\mathsf{L}.\ x \rightsquigarrow_X t_x \to \exists t'.\ t_f\, t_x \equiv t' \wedge f\, x \rightsquigarrow_Y t'}{f \rightsquigarrow_{X\to Y} t_f}$$

Note that for concrete X, the relation $\rightsquigarrow_X$ can be spelled out and defined. In Coq, we use a reified type of simple types such that the reification $\tilde{X}$ can be obtained from X using type classes. Then one can define $\rightsquigarrow_{\tilde{X}}$ by recursion on $\tilde{X}$.

For instance, for $X := (\mathbb{N} \to \mathbb{B}) \to \mathbb{L}\mathbb{N} \to \mathbb{L}\mathbb{B}$ (the type of the $\mathsf{map}_{\mathbb{N},\mathbb{B}}$ function), we have that for a closed abstraction s, the proposition $\mathsf{map}_{\mathbb{N},\mathbb{B}} \rightsquigarrow s$ is equivalent to

$$\forall f : \mathbb{N} \to \mathbb{B}.\forall t_f : \mathsf{tm}_\mathsf{L}.\ (\forall x : \mathbb{N}.\ f\,(\varepsilon_{\mathbb{N}} x) \equiv \varepsilon_{\mathbb{B}}(f\,x)) \to \forall l : \mathbb{L}\mathbb{N}.\ \ s\,(t_f)\,(\varepsilon l) \equiv \varepsilon(\mathsf{map}_{\mathbb{N},\mathbb{B}}\ f\ l).$$

The core of the certifying extraction framework is a Coq command which on input x generates a term t_x and a proof that $x \rightsquigarrow_X t_x$. The command can also be accessed via the tactic *extract*, see e.g. the *Ackermann.v* file in the accompanying Coq code. For the command to work, all auxiliary functions and constructors a function uses have to be encoded first.

For instance, to extract $\mathsf{map}_{\mathbb{N},\mathbb{B}}$ relying on the constructors [] and ::, one first has to use the command for :: obtaining $t_{::}$ and then can extract $\mathsf{map}_{\mathbb{N},\mathbb{B}}$ to the term t:

$$t_{::} := \lambda x_1 x_2 y_1 y_2.\ y_2 x_1 x_2 \qquad\qquad t := \rho(\lambda m f l.\ l\ (\varepsilon[])\ (\lambda x l'.\ t_{::}\ (f\,x)\ (m l))$$

Note that one does not need to generate $t_{[]}$, since $\varepsilon[]$ can be used instead.

27.4 Universal term

To obtain a universal term for L, we implement a step-indexed interpreter $\mathsf{eval} : \mathbb{N} \to \mathsf{tm}_\mathsf{L} \to \mathbb{O}(\mathsf{tm}_\mathsf{L})$. The defining equations follow the rule of the evaluation relation $\triangleright$:

$$\mathsf{eval}\,(\mathsf{S}\,n)\,(s\,t) = \textbf{if}\ (\mathsf{eval}\ n\ s, \mathsf{eval}\ n\ t)\ \textbf{is}\ (\mathsf{Some}\,(\lambda u), \mathsf{Some}\,t')\ \textbf{then}\ \mathsf{eval}\ n\ (u^0_{t'})\ \textbf{else}\ \mathsf{None}$$

$$\mathsf{eval}\,(\mathsf{S}\,n)\,(\lambda s)\ = \mathsf{Some}\,(\lambda s) \qquad\qquad \mathsf{eval}\ n\ t = \mathsf{None}\ \textit{in all other cases}$$

Fact 27.5. $\lambda n.\ \mathsf{eval}\ n\ s$ is stationary.

Fact 27.6. $s \triangleright t \leftrightarrow \exists n.\ \mathsf{eval}\ n\ s = \mathsf{Some}\ t$

The interpreter uses naive de Bruijn substitution, which in turn uses equality test of natural numbers. Thus when using the Coq commands of the extraction framework, one proves the following results:

Fact 27.7. The following hold:

1. There is a term t_{eq} such that $(\lambda n_1 n_2 : \mathbb{N}.\ n_1 =_{\mathbb{B}} n_2) \rightsquigarrow t_{\mathsf{eq}}$.
2. There is a term t_{subst} such that $(\lambda s n u. s_u^n) \rightsquigarrow t_{\mathsf{subst}}$.
3. There is a term t_{eval} such that $\mathsf{eval} \rightsquigarrow t_{\mathsf{eval}}$.

Lastly, we implement a term μ_{L} performing unbounded search.

Theorem 27.8. There is a closed abstraction μ_{L} such that for all closed abstraction s satisfying $\forall n : \mathbb{N}. \exists b : \mathbb{B}.\ s(\varepsilon_{\mathbb{N}} n) \equiv \varepsilon_{\mathbb{B}} b$, the following hold:

1. If $s(\varepsilon n) \equiv \varepsilon\ \mathsf{true}$, then $\exists m.\ \mu_{\mathsf{L}} s \equiv \varepsilon m$.
2. If $\mu_{\mathsf{L}} s \triangleright v$, then there exists n such that $v = \varepsilon n$, $s(\varepsilon n) \equiv \varepsilon\ \mathsf{true}$ and $\forall m < n.\ s(\varepsilon m) \equiv \varepsilon\ \mathsf{false}$.

Proof. Define $\mu_{\mathsf{L}} := \lambda s.\ \rho(\lambda r n.\ (s n)\ n\ (r(t_{\mathsf{S}}\, n)))(\varepsilon_{\mathbb{N}} 0)$. ∎

Theorem 27.9. There is a closed abstraction $\mathsf{U} : \mathsf{tm}_{\mathsf{L}}$ such that $s \triangleright t \leftrightarrow \mathsf{U}(\varepsilon_{\mathsf{tm}_{\mathsf{L}}} s) \triangleright \varepsilon_{\mathsf{tm}_{\mathsf{L}}} t$.

Proof. First obtain a term t s.t.$(\lambda s : \mathsf{tm}_{\mathsf{L}}. \lambda n : \mathbb{N}.$ **if** $\mathsf{eval}\ n\ s$ **is** $\mathsf{Some}\ v$ **then** true **else** $\mathsf{false}) \rightsquigarrow t$. Then define $\mathsf{U} := \lambda s.\ \mathsf{eval}\ (\mu_L(t s))\ s\ (\lambda x.\ x)\ (\lambda x.\ x)$. ∎

Using U, we can now prove that $\mathsf{Halt}_{\mathsf{L}}$ reduces to the halting problem of L on closed terms.

Corollary 27.10. $\mathsf{Halt}_{\mathsf{L}} \preceq_m \mathsf{Halt}'_{\mathsf{L}}$

Proof. Since U is closed by Theorem 27.9. ∎

27.5 Notions of computability theory in L

One can define notions like decidability, semi-decidability, and enumerability in terms of L. A predicate $p : X \to \mathbb{P}$ where X is L-encodable is

- L-decidable if $\exists t : \mathsf{tm}_{\mathsf{L}}.\ \mathsf{closed}\ t \wedge \forall x.\ (p x \wedge t(\varepsilon_X x) \triangleright \varepsilon_{\mathbb{B}} \mathsf{true}) \vee (\neg p x \wedge t(\varepsilon_X x) \triangleright \varepsilon_{\mathbb{B}} \mathsf{false}$.
- L-recognisable if $\exists t : \mathsf{tm}_{\mathsf{L}}.\ \mathsf{closed}\ t \wedge \forall x.\ p x \leftrightarrow \mathsf{Halt}_{\mathsf{L}}(t\ (\varepsilon_X x))$.
- L-enumerable if $\exists t : \mathsf{tm}_{\mathsf{L}}.\ \mathsf{closed}\ t \wedge \forall x.\ p x \leftrightarrow \exists n.\ t\ (\varepsilon_X n) \triangleright \varepsilon_X x$.

Note how these notions only depend on L-terms and do not mention functions.

Fundamentally, total L-terms computing a value of type Y when applied to a type X can be turned into total functions $X \to Y$. The construction uses the guarded minimisation function $\mu_{\mathbb{N}} : \forall f : \mathbb{N} \to \mathbb{B}.(\exists n. f n = \mathsf{true}) \to \mathbb{N}$ from Corollary 3.8 and the step-indexed interpreter eval.

[83] Forster and Smolka. 2017. Weak Call-by-Value Lambda Calculus as a Model of Computation in Coq.

Lemma 27.11 ([83, Lemma 27]). $\forall s.(\exists v.\ s \triangleright v) \to \Sigma v.\ s \triangleright v$

Proof. Let s be given and define $f n :=$ **if** $\mathsf{eval}\ \ n\ \ s$ **is** $\mathsf{Some}\ v$ **then** true **else** false. If $(\exists v.\ s \triangleright v)$, then also $\exists n.\ f n = \mathsf{true}$, and we have that $f(\mu_{\mathbb{N}} f) = \mathsf{true}$. Thus $s \triangleright$ **if** $\mathsf{eval}\ (\mu_{\mathbb{N}} f) s$ **is** $\mathsf{Some}\ v$ **then** v **else** $\lambda x.x$. ∎

We have discussed a similar, more abstract theorem in Section 7.6.2.
We can then show the following characterisation theorem.

Theorem 27.12. Let $X \to \mathbb{P}$ where X be L-encodable. We have that p is...

- L-decidable if and only if there is an L-computable decider $f : X \to \mathbb{B}$ for p.
- L-recognisable if and only if there is an L-computable semi-decider $f : X \to \mathbb{N} \to \mathbb{B}$ for p.
- L-enumerable if and only if there is an L-computable enumerator $f : \mathbb{N} \to \mathbb{O}X$ for p.

We can use these notions to obtain many-one reductions of problems to Halt_L.

Theorem 27.13. Let X be L-encodable and $p : X \to \mathbb{P}$ be L-recognisable. Then $p \preceq_m \mathsf{Halt}_\mathsf{L}$.

Proof. Let s_f compute f obtained via Theorem 27.12 (2). Then $\lambda x.\ \mu_\mathsf{L}(s_f\, x)$ many-one reduces p to Halt_L since

$$px \leftrightarrow (\exists n.\ f\, x\, n = \mathsf{true}) \leftrightarrow (\exists n.\ s_f\, \overline{x}\, \overline{n} \triangleright \overline{\mathsf{true}}) \leftrightarrow \mathsf{Halt}_\mathsf{L}(\mu_\mathsf{L}(s_f\, \overline{x}))$$ ∎

Since enumerable predicates on discrete types are semi-decidable, we can reduce them to the L-halting problem as well:

Theorem 27.14. Let $p : X \to \mathbb{P}$ be L-enumerable and d be an L-computable equality decider for X. Then $p \preceq_m \mathsf{Halt}_\mathsf{L}$.

Proof. Let f be an enumerator for p by Theorem 27.12 (3). The function $g\, x\, n :=$ **if** $f\, n$ **is** $\mathsf{Some}\, x'$ **then** $d\, x\, x'$ **else** false semi-decides p as in Corollary 4.57. Since f and d are L-computable, g is L-computable. By Theorem 27.13 we have $p \preceq_m \mathsf{Halt}_\mathsf{L}$. ∎

Lastly, it is often more convenient to give list enumerators, and the construction converting list-enumerators to enumerators also preserves L-computability:

Corollary 27.15. Let $p : X \to \mathbb{P}$ be list-enumerated by f (i.e. $\forall x.\ px \leftrightarrow \exists n.\ x \in f\, n$), and d be an equality decider for X. If d and f are L-computable, then $p \preceq_m \mathsf{Halt}_\mathsf{L}$.

We can now prove that the L-halting problem is L-undecidable:

Lemma 27.16. $\lambda s.\neg\mathsf{Halt}_\mathsf{L}(s(\varepsilon_{\mathsf{tm}_\mathsf{L}}\, s))$ is not L-recognisable.

Proof. Let t be an L-term satisfying $\forall s.\ \neg\mathsf{Halt}_\mathsf{L}(s(\varepsilon_{\mathsf{tm}_\mathsf{L}}\, s)) \leftrightarrow \mathsf{Halt}_\mathsf{L}(t(\varepsilon_{\mathsf{tm}_\mathsf{L}}\, s))$. Then in particular $\neg\mathsf{Halt}_\mathsf{L}(t(\varepsilon_{\mathsf{tm}_\mathsf{L}}\, t)) \leftrightarrow \mathsf{Halt}_\mathsf{L}(t(\varepsilon_{\mathsf{tm}_\mathsf{L}}\, t))$, contradiction. ∎

Corollary 27.17. $\overline{\mathsf{Halt}_\mathsf{L}}$ is not L-recognisable.

Proof. If t would recognise $\overline{\mathsf{Halt}_\mathsf{L}}$, then $\lambda x.\ t(t_{\mathsf{app}} x(t_{\varepsilon_{\mathsf{tm}_\mathsf{L}}}\, x)$, (where t_{app} computes application and $t_{\varepsilon_{\mathsf{tm}_\mathsf{L}}}$ computes $\varepsilon_{\mathsf{tm}_\mathsf{L}}$) would recognise $\lambda s.\neg\mathsf{Halt}_\mathsf{L}(s(\varepsilon_{\mathsf{tm}_\mathsf{L}}\, s))$. Contradiction. ∎

Corollary 27.18. Halt_L is L-undecidable.

We can deduce the same results for the halting problem on closed L-terms $\mathsf{Halt}'_\mathsf{L}$:

Corollary 27.19. $\overline{\mathsf{Halt}'_\mathsf{L}}$ is not L-recognisable and $\mathsf{Halt}'_\mathsf{L}$ is L-undecidable.

Proof. Let t recognise $\overline{\mathsf{Halt}'_\mathsf{L}}$. Let $t' := (\lambda x.t(t_{\mathsf{app}}(\varepsilon_{\mathsf{tm}_\mathsf{L}}\, \mathsf{U})(t_{\varepsilon_{\mathsf{tm}_\mathsf{L}}}\, x)))$. Then $\overline{\mathsf{Halt}_\mathsf{L}\, s} \leftrightarrow \overline{\mathsf{Halt}'_\mathsf{L}}(\mathsf{U}(\varepsilon_{\mathsf{tm}_\mathsf{L}}\, s)) \leftrightarrow \mathsf{Halt}_\mathsf{L}(t'(\varepsilon_{\mathsf{tm}_\mathsf{L}}\, s))$, i.e. t' recognises $\mathsf{Halt}'_\mathsf{L}$. ∎

We can lift Halt_L to a relation of type $\mathbb{N}\to\mathbb{N}\to\mathbb{P}$ by defining $R_{\mathsf{Halt}_\mathsf{L}}\,nm := m = 0 \leftrightarrow \exists s.\, en = \mathsf{Some}\, s \wedge \mathsf{Halt}_\mathsf{L}\, s$, where e is an enumerator of tm_L.

Corollary 27.20. The relation $R_{\mathsf{Halt}_\mathsf{L}} : \mathbb{N}^1 \to \mathbb{N} \to \mathbb{P}$ is not L-computable.

27.6 Related work

[175] Myreen and Owens. 2014. Proof-producing translation of higher-order logic into pure and stateful ML.

[145] Kumar, Myreen, Norrish, and Owens. 2014. CakeML.

[118] Hupel and Nipkow. 2018. A Verified Compiler from Isabelle/HOL to CakeML.

[173] Mullen, Pernsteiner, Wilcox, Tatlock, and Grossman. 2018. Œuf: minimizing the Coq extraction TCB.

[2] Anand, Appel, Morrisett, Paraskevopoulou, Pollack, Belanger, Sozeau, and Weaver. 2017. CertiCoq: A verified compiler for Coq.

[156] Letouzey. 2004. Programmation fonctionnelle certifiée: l'extraction de programmes dans l'assistant Coq.

[140] Köpp. 2018. Automatically verified program extraction from proofs with applications to constructive analysis.

Myreen and Owens [175] implement a proof-producing translation from the higher-order logic implemented in the HOL4 system with a state-and-exception monad into CakeML [145]. The translation also produces proofs for the translated terms, similar to our approach. Hupel and Nipkow [118] give a verified compiler from a deep embedding of Isabelle/HOL to CakeML. Similar to our work, they use a logical relation to connect Isabelle definitions to an intermediate representation.

Mullen et al. [173] provide a verified compiler from a subset of Coq to assembly. Anand et al. [2] report on ongoing work on verifying a compiler for Coq to assembly, also based on the MetaCoq framework. They first compile Coq functions into Clight, an intermediate language of the CompCert compiler. Letouzey [156] describes the theoretical foundations of extraction in Coq. Our logical relation can be seen as a light-weight version of his simulation predicate for simple polymorphic types.

Köpp [140] verifies program extraction for functions in the Minlog proof assistant into a λ-calculus-like system and obtains a certifying extraction by instantiating the correctness theorem.

CHAPTER 28

Equivalence proofs

We here explain two techniques how to simulate other models of computation in L, and two techniques to prove many-one equivalences, respectively.

First, we show that every finite function, i.e. every function where the domain is a finite discrete type, is L-computable. Since step-functions of Turing machines are finite functions, they are always L-computable. We can use this to obtain a compilation of Turing machines to L by looping the step-function.

Secondly, we show that TM-computable relations are L-computable. Until now, we have glossed over the details of such proofs, but here now exemplarily show how pre- and post-processing of input and output works.

Thirdly, we show that L can simulate μ-recursive functions. The challenge here is that μ-recursive functions are defined using a dependent inductive type, and that the certifying extraction framework does not support functions on it. We thus introduce a novel general technique of identifying the *shape of a type*, on which functions can be defined which are extractable to L.

Lastly, we employ the results from the last chapter to show that PCP and minimal provability in first-order logic both reduce to $\mathsf{Halt}_{\mathsf{L}}$.

$\mathsf{Halt}_{\mathsf{L}}$
→ Sec. 11.1, Page 109

Publications Sections 28.1 and 28.2 contain adapted pieces of text from [76], and section 28.3 contains adapted pieces of text from [153], which were all written solely by the author of this thesis. Section 28.5 is based on [74].

[76] Forster and Kunze. "A certifying extraction with time bounds from Coq to call-by-value λ-calculus." *International Conference on Interactive Theorem Proving*. 2019.

[153] Larchey-Wendling and Forster. "Hilbert's Tenth Problem in Coq (extended version)." *arXiv preprint arXiv:2003.04604*. 2020.

[74] Forster, Kirst, and Wehr. "Completeness Theorems for First-Order Logic Analysed in Constructive Type Theory" *International Symposium on Logical Foundations of Computer Science*

28.1 Simulating Turing machines

To simulate Turing machines in L, we first give an alternative, executable semantics for Turing machines based on iteration of a step-function. Secondly, we implement a (potentially non-terminating) iteration combinator in L.

We define a function $\mathsf{nxt}_M : Q_M \times \mathsf{tp}_\Sigma^n \to (Q_M \times \mathsf{tp}_\Sigma^n) + \mathsf{tp}_\Sigma^n$ and a polymorphic function $\mathsf{loop} : (X \to X + Y) \to X \to \mathbb{N} \to \mathbb{O} Y$ as follows:

$$\mathsf{nxt}_M(q,t) := \textbf{if } \mathsf{halt} q \textbf{ then } \mathsf{inr}\, t$$
$$\textbf{else let } (q', a) := \delta_M(q, \mathsf{curr}\, t) \textbf{ in}$$
$$\mathsf{inl}\,(q', \mathsf{map}_2(\lambda(c,m)t.\mathsf{mv}\, m\,(\mathsf{wr}\, c\, t))\, a\, t)$$

$$\mathsf{loop}\, f\ x\, 0 := \mathsf{None} \qquad \mathsf{loop}\, f\ x\, (\mathsf{S}\, n) := \mathsf{loop}\, f\ x'\, n\ (\text{if } f\, x = \mathsf{inl}\, x')$$

$$\mathsf{loop}\, f\ x\, (\mathsf{S}\, n) := \mathsf{Some}\, y\ (\text{if } f\, x = \mathsf{inr}\, y)$$

Fact 28.1. $(\exists i.\ \mathsf{loop}\, \mathsf{nxt}_M\, (q_0, t)\, i = \mathsf{Some}\, t') \leftrightarrow \exists q'.\ M(q,t) \triangleright (q', t')$

Theorem 28.2. Let X be finite and L-encodable, d_X decide equality on X and be L-computable, and Y be L-encodable. Then any function $f : X \to Y$ is L-computable.

Proof. Let l be a list containing all elements of X. If $l = []$, i.e. there are no elements in X, the claim is trivial (via the term $\lambda x.x$.). If l contains at least the element $x_0 : X$, we define

$$g[] := f\, x_0 \qquad g((x', y') :: l)x := \mathbf{if}\ d_X x x'\ \mathbf{then}\ y'\ \mathbf{else}\ g\, x\, l$$

Clearly, g is L-computable by a term t_g. We furthermore have that $f\, x = g\, [(x, f\, x) \mid x \in l]\, x$. Thus, f is L-computable by $t_g\ (\varepsilon_{\mathbb{L}(X \times Y)}\, [(x, f\, x) \mid x \in l])$. ∎

Corollary 28.3. Let $M : \mathsf{TM}^n_\Sigma$. There is $s_{\mathsf{nxt}_M} : \mathsf{tm}_\mathsf{L}$ such that $s_{\mathsf{nxt}_M} \overline{(q,t)} \triangleright \overline{\mathsf{nxt}(q,t)}$.

We define a term s_{loop} which loops a given function f on a given value x until a value y is found, or indefinitely if not. Since the extraction framework only covers total functions we manually implement s_{loop}, relying on the recursion combinator ρ:

$$s_{\mathsf{loop}} := \rho(\lambda r f x.\, f\, x (\lambda x' z.\, r f\, x')(\lambda y z.\, y)(\lambda z.\, z))$$

Note that we could have implemented s_{loop} by using μ_L, which would however be inefficient. This simulation we use has linear overhead, but we do not prove this result formally.

Lemma 28.4. Let f be computable by s_f, i.e. $\forall x.\, s_f\, \overline{x} \triangleright \overline{f\, x}$. We have the following:

1. If $\mathsf{loop}\, f\ x\, i = \mathsf{Some}\, y$, then $s_{\mathsf{loop}}\, s_f\ \overline{x}\, \overline{i} \triangleright \overline{y}$.
2. If $s_{\mathsf{loop}}\, s_f\ \overline{x}$ terminates, there exist i and y such that $\mathsf{loop}\, f\ x\, i = \mathsf{Some}\, y$.

This suffices to prove the simulation theorem.

Theorem 28.5. There is $s_{\mathsf{sim}} : \mathsf{tm}_\mathsf{L}$ such that for all $M : \mathsf{TM}^n_\Sigma$ and $t : \mathsf{tp}^n_\Sigma$:

1. If $M(q_0, t) \triangleright (q, t')$, then $s_{\mathsf{sim}}\ s_{\mathsf{nxt}_M}\ \overline{t} \triangleright \overline{t'}$.
2. If $s_{\mathsf{sim}}\ s_{\mathsf{nxt}_M}\ \overline{t} \triangleright v$, then $\exists q t'.\, M(q_0, t) \triangleright (q, t')$.

28.2 TM-computable relations are L-computable

Let $R : \mathbb{N}^k \to \mathbb{N} \to \mathbb{P}$ be computable by $M : \mathsf{TM}^n_\Sigma$. We define an L-term computing R by taking $n_1, \ldots, n_k$ as input, converting them to their respective TM-encoding, and then running M with the help of s_{sim}. For conciseness, we define $\mathsf{s}^n := \underbrace{[\mathsf{s}, \ldots, \mathsf{s}]}_{n \text{ times}}$. Step-by-step, s has to:

1. Expect input in the form $s\, \overline{n_1} \ldots \overline{n_k}$,
2. for $1 \le i \le k$ compute $t_i := \overline{\mathsf{midtp}\, []\, \mathsf{bl}\, \mathsf{s}^{n_i}}$, i.e. the L-encoding of the TM-encoding of n_i.
3. run the simulation $s_{\mathsf{sim}}\ s_{\mathsf{nxt}_M}\ \overline{(\mathsf{niltp}, t_1, \ldots, t_k, \mathsf{niltp}, \ldots, \mathsf{niltp})}$.
4. this computation will (if it terminates) terminate with a value $\overline{(\mathsf{midtp}\, []\, b\, \mathsf{s}^m, t'_2 \ldots, t'_n)}$,
5. meaning s has to output $\overline{m}$.

Three challenges arise: the term s has to be defined parametric in k, the L-encoding of the numbers $n_1, \dots, n_k$ has to be converted to the L-encoding $\overline{[\mathsf{niltp}, t_1, \dots, t_k, \mathsf{niltp}, \dots, \mathsf{niltp}]}$, and the L encoding of a result t' has to be analysed, and the TM-encoding of a number m contained in $t'[0]$ has to be converted to the L-encoding of m.

For the first task, we implement k-ary substitutions and combinators.

Fact 28.6. One can define functions $s_u^n : \mathsf{tm}_L$ where $s : \mathsf{tm}_\mathsf{L}, n : \mathbb{N}, u : \mathsf{tm}_\mathsf{L}^k$ and

$$\lambda_k : \mathsf{tm}_\mathsf{L} \to \mathsf{tm}_\mathsf{L} \qquad \mathsf{app}_k : \mathsf{tm}_\mathsf{L} \to \mathsf{tm}_\mathsf{L}^k \to \mathsf{tm}_\mathsf{L} \qquad \mathsf{vars}_k : \mathsf{tm}_L^k$$

such that the following hold:

1. $\mathsf{vars}_{\mathsf{S}\,k} = k :: \mathsf{vars}_k$,
2. $(\mathsf{app}_k s(s_1, \dots, s_k))_u^n = \mathsf{app}_k(s_u^n)((t_1)_u^n, \dots, (t_k)_u^n)$,
3. if all elements of u are closed abstractions, then $\mathsf{app}_k(\lambda_k s)u \succ^k s_u^0$.

The second and third tasks can again be done by extraction.

Fact 28.7. There is a closed abstraction s_{prep} such that $s_{\mathsf{prep}}\,\overline{(n_1, \dots, n_k)} \triangleright \overline{[\mathsf{niltp}, t_1, \dots, t_k, \mathsf{niltp}, \dots, \mathsf{niltp}]}$, where $t_i := \mathsf{midtp}\,[\,]\,\mathsf{bl}\,\mathsf{s}^{n_i}$.

Fact 28.8. There is a closed abstraction $s_{\mathsf{unenc}_\mathsf{TM}}$ such that if $t[0] = \mathsf{midtp}\,[\,]\,\mathsf{bl}\,\mathsf{s}^m$ we have $s_{\mathsf{unenc}_\mathsf{TM}}\,\overline{t} \triangleright \overline{\mathsf{Some}\,m}$ and $s_{\mathsf{unenc}_\mathsf{TM}}\,\overline{t} \triangleright \overline{\mathsf{None}}$ otherwise.

Theorem 28.9. TM-computable relations $R : \mathbb{N}^k \to \mathbb{N} \to \mathbb{N}$ are L-computable.

Proof. Define $s_M := \lambda_k.s_{\mathsf{unenc}_\mathsf{TM}}(s_{\mathsf{sim}}\; s_{\mathsf{nxt}_M}\; (s_{\mathsf{prep}}(s_{\mathsf{cons}}k(\dots(s_{\mathsf{cons}}\,0\,\overline{0})))))(\lambda x.x)\,\overline{0}$. ∎

Corollary 28.10. $\mathsf{Halt}_\mathsf{TM} \preceq_m \mathsf{Halt}_\mathsf{L}$

Corollary 28.11. The relation $R_{\mathsf{Halt}_\mathsf{L}} : \mathbb{N}^1 \to \mathbb{N} \to \mathbb{P}$ is not TM-computable.

$R_{\mathsf{Halt}_\mathsf{L}}$ → Sec. 27.5, Page 182

28.3 Simulating μ-recursive functions

To simulate μ-recursive functions in L we implement a step-indexed interpreter. Recall that the type func_k of μ-recursive functions is heavily dependent: It has a type index in k, makes use of finite types $\mathbb{F}_k$, and has a nested use of the dependent vector type, i.e. contains subterms of type $(\mathsf{func}_k)^i$. Since the extraction from Coq to L does not support dependent or nested types, a direct step-indexed interpreter will not be extractable. Thus, we use a general technique and implement a step-indexed interpreter working on the **shape** of μ-recursive functions.

Def. shape

A shape for a type I mirrors the constructors of I, but without any dependent types. If I furthermore has nested applications of types $N_1, \dots, N_n$, the shape also contains the constructors of $N_1, \dots, N_n$. That is, if I has i constructors and $N_1, \dots, N_n$ have $m_1, \dots, m_n$ constructors respectively, the shape of I has $i + m_1 + \dots + m_n$ constructors. The type shape of the type func is defined as in Figure 28.1.

Note that we re-use constructor names. In the shape we have $j : \mathbb{N}$ instead of $j : \mathbb{F}_k$ for proj, $g : \mathsf{shape}$ instead of $G : (\mathsf{func}_k)^i$ for comp, and the constructors cons and nil as the constructors of the vector type are added.

It is straightforward to implement mutually recursive functions $\mathsf{erase} : \mathsf{func}_k \to \mathsf{shape}$ and $\mathsf{erase}' : (\mathsf{func}_k)^i \to \mathsf{shape}$ which are essentially the identity. We call a shape $s : \mathsf{shape}$ **valid**

$$\frac{c : \mathbb{N}}{\mathsf{cst}\, c : \mathsf{shape}} \qquad \frac{}{\mathsf{zero} : \mathsf{shape}} \qquad \frac{}{\mathsf{succ} : \mathsf{shape}} \qquad \frac{j : \mathbb{N}}{\mathsf{proj}\, j : \mathsf{shape}}$$

$$\frac{f : \mathsf{shape} \qquad g : \mathsf{shape}}{\mathsf{comp}\, f\ G : \mathsf{shape}} \qquad \frac{f : \mathsf{shape} \qquad g : \mathsf{shape}}{\mathsf{primrec}\, f\ g : \mathsf{shape}} \qquad \frac{f : \mathsf{shape}}{\mu f : \mathsf{shape}}$$

$$\frac{f : \mathsf{shape} \qquad g : \mathsf{shape}}{\mathsf{cons}\ f\ g : \mathsf{shape}} \qquad \frac{}{\mathsf{nil} : \mathsf{shape}}$$

$$\begin{aligned}
[\![\mathsf{cst}\, c]\!]^m_{\mathsf{S}\, i} l &:= \mathsf{Some}\,(\mathsf{inl}\, c) \\
[\![\mathsf{zero}]\!]^m_{\mathsf{S}\, i} l &:= \mathsf{Some}\,(\mathsf{inl}\, 0) \\
[\![\mathsf{succ}]\!]^m_{\mathsf{S}\, i} (x :: l) &:= \mathsf{Some}\,(\mathsf{inl}\,(1 + x)) && \textit{if } l[n] = \mathsf{Some}\, x \\
[\![\mathsf{proj}\, j]\!]^m_{\mathsf{S}\, i} l &:= \mathsf{Some}\,(\mathsf{inl}\, x) && \textit{if } l[j] = \mathsf{Some}\, x \\
[\![\mathsf{comp}\, f\ g]\!]^m_{\mathsf{S}\, i} l &:= \mathsf{Some}\,(\mathsf{inl}\, x) && \textit{if } [\![g]\!]^m_i = \mathsf{Some}\,(\mathsf{inr}\, l') \textit{ and } [\![f]\!]^m_i l' = \mathsf{Some}\,(\mathsf{inl}\, x) \\
[\![\mathsf{primrec}\, f\ g]\!]^m_{\mathsf{S}\, i} (0 :: l) &:= \mathsf{Some}\,(\mathsf{inl}\, x) && \textit{if } [\![f]\!]^m_i l = \mathsf{Some}\,(\mathsf{inl}\, x) \\
[\![\mathsf{primrec}\, f\ g]\!]^m_{\mathsf{S}\, i} (\mathsf{S}\, n :: l) &:= \mathsf{Some}\,(\mathsf{inl}\, x) && \textit{if } [\![\mathsf{primrec}\, f\ g]\!]^m_i (n :: l) = \mathsf{Some}\,(\mathsf{inl}\, x) \textit{ and } [\![g]\!]^m_i (n :: y :: l) = \mathsf{Some}\,(\mathsf{inl}\, x) \\
[\![\mu f]\!]^m_{\mathsf{S}\, i} l &:= \mathsf{Some}\,(\mathsf{inl}\, m) && \textit{if } [\![f]\!]^0_c (m :: l) = \mathsf{Some}\,\mathsf{inl}\, 0 \\
[\![\mu f]\!]^m_{\mathsf{S}\, i} l &:= \mathsf{Some}\,(\mathsf{inl}\, x) && \textit{if } [\![f]\!]^0_c (m :: l) = \mathsf{Some}\,\mathsf{inr}\,(\mathsf{S}\, y) \textit{ and } [\![\mu f]\!]^{\mathsf{S}\, m}_i l = \mathsf{Some}\,(\mathsf{inl}\, x) \\
[\![\mathsf{nil}]\!]^m_{\mathsf{S}\, i} l &:= \mathsf{Some}\,(\mathsf{inr}\,[\,]) \\
[\![\mathsf{cons}\, f\ g]\!]^m_{\mathsf{S}\, i} l &:= \mathsf{Some}\,(\mathsf{inr}\,(x :: l')) && \textit{if } [\![f]\!]^m_c l = \mathsf{Some}\,(\mathsf{inl}\, x) \textit{ and } [\![g]\!]^m_i l = \mathsf{Some}\,(\mathsf{inr}\, l') \\
[\![f]\!]^m_i l &:= \mathsf{None} && \textit{in all other cases}
\end{aligned}$$

Figure 28.1.: Shape and step-indexed interpreter of μ-recursive functions.

if it corresponds to a μ-recursive function or a vector of μ-recursive functions, i.e. if $\exists k{:}\,\mathbb{N}.\ (\exists f : \mathsf{func}_k.\ s = \mathsf{erase} f) \vee (\exists i{:}\,\mathbb{N}.\exists l{:}\,(\mathsf{func}_k)^i.\ s = \mathsf{erase}' l)$.

For shape we can now define a step-indexed evaluation function

$$[\![\cdot]\!]{:}\,\mathsf{shape} \to \mathbb{N} \to \mathbb{N} \to \mathbb{L}\mathbb{N} \to \mathbb{O}(\mathbb{N} + \mathbb{L}\mathbb{N}).$$

A call $[\![f]\!]_i^m l$ uses i as step-index, m as auxiliary counter to implement unbounded search, and l as input. If $[\![f]\!]_i^m l = \mathsf{Some}(\mathsf{inl}\, v)$, then v is the value of the evaluation. If $[\![f]\!]_i^m l = \mathsf{Some}(\mathsf{inr}\, l')$, then f encoded a list of functions via cons and nil which pointwise evaluated to $l'{:}\,\mathbb{L}\mathbb{N}$. If $[\![f]\!]_i^m l = \mathsf{None}$ either as usual the step-index c was not big enough, or the function f does not terminate on input l, or f is not a valid shape. The definition of $[\![f]\!]_i^m$ is displayed in Figure 28.1.

Lemma 28.12. For all $f : \mathsf{func}_k$ and $v{:}\,\mathbb{N}^k$ we have $f[v] \triangleright n \leftrightarrow \exists i.\ [\![f]\!]_i^0 v = \mathsf{Some}(\mathsf{inl}\, n)$.

Proof. By first defining a predicate $f[v] \triangleright^i n$ s.t. $f[v] \triangleright n \leftrightarrow \exists i.\ f[v] \triangleright^i n$ and then proving $f[v] \triangleright^i n \leftrightarrow [\![f]\!]_i^0 v = \mathsf{Some}(\mathsf{inl}\, n)$ by complete induction on i. ∎

Fact 28.13. The function $[\![\cdot]\!]{:}\,\mathsf{shape} \to \mathbb{N} \to \mathbb{N} \to \mathbb{L}\mathbb{N} \to \mathbb{O}(\mathbb{N} + \mathbb{L}\mathbb{N})$ is L-computable by a term t_{int}.

Fact 28.14. For every f, the function $\lambda i.$**if** $[\![f]\!]_i^0 v$ **is** $\mathsf{Some}(\mathsf{inl}\, n)$ **then** true **else** false is L-computable by a term $t_{\mathsf{test}}(f, v)$.

Theorem 28.15. For all $f : \mathsf{func}_k$ and $v{:}\,\mathbb{N}^k$ we have

$$f[v] \triangleright n \leftrightarrow t_{\mathsf{sim}}(\mu_{\mathsf{L}}(t_{\mathsf{test}}(f, v)))(\varepsilon(\mathsf{erase} f))(\varepsilon v) \triangleright \varepsilon n.$$

Corollary 28.16. $\mathsf{Halt}_\mu \preceq_m \mathsf{Halt}_{\mathsf{L}}$

Since there is no pre- and post-processing of input necessary, we immediately also obtain:

Corollary 28.17. Every μ-recursive relation $R{:}\,\mathbb{N}^k \to \mathbb{N} \to \mathbb{P}$ is L-computable.

Corollary 28.18. The relation $R_{\mathsf{Halt}_{\mathsf{L}}} : \mathbb{N}^1 \to \mathbb{N} \to \mathbb{P}$ is not μ-recursive.

28.4 Semi-deciding PCP

We construct a semi-decider for BPCP and extract it to L. Recall that

$$\mathsf{BPCP}(R{:}\,\mathbb{L}(\mathbb{L}\mathbb{B} \times \mathbb{L}\mathbb{B})) := \exists A{:}\,\mathbb{L}(\mathbb{L}\mathbb{B} \times \mathbb{L}\mathbb{B}).\ A \neq [\,] \wedge A \subseteq R \wedge \tau_1 A = \tau_2 A$$

Since the condition after the existential quantification is decidable, a semi-decider can take as input $n{:}\,\mathbb{N}$, obtain A via an enumerator for $\mathbb{L}(\mathbb{L}\mathbb{B} \times \mathbb{L}\mathbb{B})$ and then decide the condition for A.

We thus need the following, which are straightforward:

Fact 28.19. There is an L-computable ...

1. equality decider d_{eq} for $\mathbb{L}\mathbb{B}$.
2. enumerator e for $\mathbb{L}(\mathbb{L}\mathbb{B} \times \mathbb{L}\mathbb{B})$.
3. decider d_{sub} for $\lambda AR.\ A \subseteq R$.

Furthermore we have:

Fact 28.20. τ_1 and τ_2 are L-computable.

We can thus plug these functions together to obtain:

Theorem 28.21. BPCP is semi-decided by the following L-computable function:

$$f\,Rn := \textbf{if } e_{\mathbb{L}\mathbb{B}}n \textbf{ is } \mathsf{Some}(c :: A) \textbf{ then } d_{\mathsf{sub}}(c :: A)R \wedge_{\mathbb{B}} d_{\mathsf{eq}}(\tau_1(c :: A))(\tau_2(c :: A)) \textbf{ else } \mathsf{false}$$

Corollary 28.22. $\mathsf{BPCP} \equiv_m \mathsf{Halt}_{\mathsf{L}}$

28.5 Enumerating first-order logic

We construct a parametric list enumerator for the minimal natural deduction system and extract it to L. The structure of the natural deduction systems seems to make an enumerability proof easier than a semi-decidability proof, and in general list enumerators are easier to construct than enumerators. The constructed list enumerator e will be parametric in the sense that $\Gamma \vdash \varphi \leftrightarrow \exists n.\, \varphi \in e_\Gamma n$.

We again need some results which are automatic to prove using the certifying extraction framework:

Fact 28.23. There is an L-computable ...

1. equality decider d_{tm} for tm.
2. enumerator e_{tm} for tm.
3. equality decider d_{fm} for fm.
4. enumerator e_{fm} for fm.

We then provide a parameterised list enumerator for minimal natural deduction:

Lemma 28.24. There is an L-computable, cumulative, parametric list-enumerator $e : \mathbb{L}(\mathsf{fm}) \to \mathbb{N} \to \mathbb{O}(\mathsf{fm})$ of minimal natural deduction, i.e. $\forall \Gamma \varphi.\, \Gamma \vdash \varphi \leftrightarrow \exists n.\, \varphi \in e_\Gamma n$.

Proof. The following definition of e_Γ does the job:

$$e_\Gamma 0 := \Gamma \qquad e_\Gamma(\mathsf{S}\, n) := \left[\varphi \dot{\to} \psi \mid \psi \in e_{\varphi :: \Gamma} n\right] +\!\!+ [\psi \mid \psi \in e_{\mathsf{fm}} n, \varphi \in e_\Gamma n, (\varphi \dot{\to} \psi) \in e_\Gamma n] \qquad \blacksquare$$

Note that providing enumerators for intuitionistic and classical natural deduction is also easy, but since they many-one reduce to minimal provability via double-negation and A-translation, there is no need.

Theorem 28.25. There is an L-computable list-enumerator for minimal provability, i.e. $\lambda \varphi.\, [] \vdash \varphi$.

Corollary 28.26. $\lambda \varphi.\, [] \vdash \varphi \equiv_m \mathsf{Halt}_{\mathsf{L}}$

We can now compose the two corollaries to prove the many-one equivalence of BPCP and first-order provability. Note that this is a *synthetic* many-one equivalence not mentioning L, which was however easy to prove by using L and the certifying extraction internally.

Corollary 28.27. $\mathsf{BPCP} \equiv_m \lambda \varphi.\, [] \vdash \varphi$

CHAPTER 29

CT in L

In Chapter 7 we discussed the constructivist axiom CT stating that every function $f : \mathbb{N} \to \mathbb{N}$ is L-computable, but without defining CT formally.

In this final chapter, we tie loose ends regarding CT. We define $\phi_{\mathsf{L}} : \mathbb{N} \to \mathbb{N} \to \mathbb{ON}$ universal for all L-computable functions $\mathbb{N} \to \mathbb{N}$ and define CT_{L}. We then give characterisations of CT_{L} in terms of boolean functions $\mathbb{N} \to \mathbb{B}$ and semi-decidable, enumerable, and decidable predicates. Using the results from Part II, we show that CT can equivalently be defined using the other models of computation. We also prove that CT_{L} implies $\mathcal{U}p \to \neg\mathcal{D}p$ as motivated in Part III. Lastly, we prove the S_n^m theorem, necessary to deduce SCT from CT as motivated in Part I, and discuss a possible mechanised admissibility proof for CT in CIC.

Publications All results are novel.

29.1 CT for L

Recall that we introduced the axiom CT_ϕ for $\phi : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \to \mathbb{ON}$ as follows:

CT_ϕ → Sec. 6.1, Page 54

$$\mathsf{CT}_\phi := \forall f : \mathbb{N} \to \mathbb{N}. \exists c : \mathbb{N}. \forall x : \mathbb{N}. \exists n : \mathbb{N}.\ \phi_c^n x = \mathsf{Some}\,(f\,x)$$

In Section 27.4 we introduced a step-indexed interpreter $\mathsf{eval} : \mathsf{tm}_{\mathsf{L}} \to \mathbb{N} \to \mathbb{O}\mathsf{tm}_{\mathsf{L}}$. We here use eval to implement $\phi : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \to \mathbb{ON}$.

To do so, we first verify a retraction (R, I) of closed terms to $\mathbb{N}$, i.e. $(\mathsf{closed}\, s \land \exists t.\ s = \lambda t) \leftrightarrow R(Is) = \mathsf{Some}\, s$, and an inverse function unenc of $\varepsilon_{\mathbb{N}}$, i.e. $\mathsf{unenc}(\varepsilon_{\mathbb{N}} n) = \mathsf{Some}\, n$.

retraction
→ Sec. A.4, Page 209

Fact 29.1. There is a function $\mathsf{unenc} : \mathsf{tm}_{\mathsf{L}} \to \mathbb{ON}$ such that $\mathsf{unenc}(\varepsilon_{\mathbb{N}} n) = \mathsf{Some}\, n$.

Fact 29.2. The type of L-terms tm_L is discrete and enumerable.

Lemma 29.3. There are functions $R : \mathbb{N} \to \mathbb{O}(\mathsf{tm}_{\mathsf{L}})$ and $I : \mathsf{tm}_{\mathsf{L}} \to \mathbb{N}$ such that $\mathsf{closed}\, s \leftrightarrow R(Is) = \mathsf{Some}\, s$.

Proof. Since tm_{L} is discrete and enumerable by Fact 29.2, we obtain a retraction (I, R') from tm_{L} to $\mathbb{N}$ by Corollary 4.34, i.e. $\forall s.\ R'(Is) = \mathsf{Some}\, s$. Let d_{closed} decide closedness of terms. Pick (I, R) with $Rn := \textbf{if}\ R'n\ \textbf{is}\ \mathsf{Some}\, t\ \textbf{then if}\ d_{\mathsf{closed}}\, t\ \textbf{then}\ \mathsf{Some}\, t\ \textbf{else}\ \mathsf{None}\ \textbf{else}\ \mathsf{None}$. ∎

For an application $\phi_c^n x$ we translate the code c to a term t obtained from Rc, and then evaluate the application $t\ (\varepsilon_{\mathbb{N}} n)$ for n steps using the step-indexed interpreter eval:

$$\phi_c^n x := \textbf{if}\ Rc\ \textbf{is}\ \mathsf{Some}\, t\ \textbf{then if}\ \mathsf{eval}\ (t\ (\varepsilon_{\mathbb{N}} n))\ n\ \textbf{is}\ \mathsf{Some}\, v\ \textbf{then}\ \mathsf{unenc}\ v\ \textbf{else}\ \mathsf{None}\ \textbf{else}\ \mathsf{None}$$

We define

$$\mathsf{CT}_{\mathsf{L}} := \mathsf{CT}_\phi$$

for ϕ defined using eval as above.

Similar to how SCT is equivalent to EA, we can prove that CT_L can equivalently be phrased employing various other notions in L.

Fact 29.4. There are t_{embed} and t_{unembed} computing $\lambda\langle n,m\rangle.(n,m)$ and $\lambda nm.\langle n,m\rangle$, respectively.

$\langle n,m\rangle$
→ Sec. A.1, Page 207

Theorem 29.5. The following are equivalent:

1. CT_L
2. $\forall f:\mathbb{N}\to\mathbb{N}.\ \exists t:\mathsf{tm}_\mathsf{L}.\ \mathsf{closed}\, t \land \forall x:\mathbb{N}.\ t\ (\varepsilon_{\mathbb{N}} x) \triangleright \varepsilon_{\mathbb{N}}(f\,x)$
3. $\forall f:\mathbb{N}\to\mathbb{B}.\ \exists t:\mathsf{tm}_\mathsf{L}.\ \mathsf{closed}\, t \land \forall x:\mathbb{N}.\ t\ (\varepsilon_{\mathbb{N}} x) \triangleright \varepsilon_{\mathbb{B}}(f\,x)$
4. $\forall p:\mathbb{N}\to\mathbb{P}.\ \mathcal{S}p \to p$ *is* L*-recognisable*
5. $\forall p:\mathbb{N}\to\mathbb{P}.\ \mathcal{E}p \to p$ *is* L*-enumerable*
6. $\forall p:\mathbb{N}\to\mathbb{P}.\ \mathcal{D}p \to p$ *is* L*-decidable*

Proof. (1)↔(2) is by using the correctness theorem of eval. (6)↔(3) and (2)→(3) are trivial.

It thus suffices to prove an implication chain (3)→(4)→(5)→(2):

(3) → (4): Let f be a semi-decider f of p. Obtain t for $\lambda\langle x,n\rangle.\ f\,x\,n$ via (2). Now $\lambda xn.\ t\ (t_{\mathsf{embed}} x n)$ computes f.

(4) → (5): Let p be enumerable. Then it is also semi-decidable by Corollary 4.57. By (4), we have an L-computable semi-decider f for p. Since the construction of Fact 4.54 is L-computable, p is L-enumerable.

(5) → (2): Let f be given. Then $\lambda\langle x,y\rangle.\ f\,x = y$ is enumerable. Thus via (5) there is g enumerating it, with t_g computing g. Then unbounded search μ_L allows searching for the value of x in the co-domain of t_g. ∎

We can then deduce that $\mathcal{U}p$ is indeed a synthetic characterisation of undecidability:

Theorem 29.6. Let $p:X\to\mathbb{P}$ and CT_L. Then $\mathcal{U}p \to \neg\mathcal{D}p$.

Proof. Assume $\mathcal{U}p$ and $\mathcal{D}p$. Thus by definition we have $\mathcal{E}(\overline{\mathsf{Halt}_{\mathsf{TM}_1}})$. Since $\overline{\mathsf{Halt}_{\mathsf{TM}_1}} \equiv_m \overline{\mathsf{Halt}'_\mathsf{L}}$ by Corollary 28.10 we have $\mathcal{E}(\overline{\mathsf{Halt}'_\mathsf{L}})$. Since tm_L is discrete we have $\mathcal{S}(\overline{\mathsf{Halt}'_\mathsf{L}})$ by Corollary 4.57.

By use of CT_L and Theorem 29.5 (4) we have that $(\overline{\mathsf{Halt}'_\mathsf{L}}$ is L-recognisable. Contradiction to Corollary 27.19. ∎

Using the equivalence proofs from Part II and Section 28.3, we have:

Corollary 29.7. The following are equivalent:

1. CT_L
2. $\forall f:\mathbb{N}\to\mathbb{N}.\ (\lambda xv.\ f\,x = v)$ *is* L*-computable*
3. $\forall f:\mathbb{N}\to\mathbb{N}.\ (\lambda xv.\ f\,x = v)$ *is* TM*-computable*
4. $\forall f:\mathbb{N}\to\mathbb{N}.\ (\lambda xv.\ f\,x = v)$ *is* BSM*-computable*
5. $\forall f:\mathbb{N}\to\mathbb{N}.\ (\lambda xv.\ f\,x = v)$ *is* CM*-computable*
6. $\forall f:\mathbb{N}\to\mathbb{N}.\ (\lambda xv.\ f\,x = v)$ *is* FRACTRAN*-computable*
7. $\forall f:\mathbb{N}\to\mathbb{N}.\ (\lambda xv.\ f\,x = v)$ *is Diophantine*
8. $\forall f:\mathbb{N}\to\mathbb{N}.\ (\lambda xv.\ f\,x = v)$ *is* μ*-recursively computable*

Assuming CT_L implies an identification of synthetic definitions with definitions in L:

Fact 29.8. Assuming CT_L we have for all $p : \mathbb{N} \to \mathbb{P}$:

1. $\mathcal{D}p \leftrightarrow \exists t : \mathsf{tm}_\mathsf{L}.\ \mathsf{closed}\ t \wedge \forall n.\ (pn \wedge t(\varepsilon_\mathbb{N} n) \triangleright \varepsilon_\mathbb{B}\mathsf{true}) \vee (\neg pn \wedge t(\varepsilon_\mathbb{N} n) \triangleright \varepsilon_\mathbb{B}\mathsf{false})$
2. $\mathcal{S}p \leftrightarrow \exists t : \mathsf{tm}_\mathsf{L}.\ \mathsf{closed}\ t \wedge\ \forall n.\ pn \leftrightarrow \mathsf{Halt}_\mathsf{L}(t(\varepsilon_\mathbb{N} n))$
3. $\mathsf{MP} \leftrightarrow \forall t : \mathsf{tm}_\mathsf{L}.\ \neg\neg(\mathsf{Halt}_\mathsf{L}\, t) \to \mathsf{Halt}_\mathsf{L}\, t$
4. $\mathcal{E}p \leftrightarrow p \preceq_m \mathsf{Halt}_\mathsf{L}$

By Theorem 29.5, (1) and (2) are equivalent to CT_L. (3) is clearly strictly weaker, since it is implied by LEM, but LEM does not imply CT_L. (4) is weaker as well: One needs to know that the many-one reduction is L-computable to conclude CT_L.

29.2 The S_n^m theorem

Recall the axiom SMN_ϕ stating the $m = n = 1$ case of the S_n^m theorem, which implies the general case:

$$\mathsf{SMN}_\phi := \Sigma\sigma : \mathbb{N} \to \mathbb{N} \to \mathbb{N}.\forall cxyv.\ (\exists n.\ \phi^n_{\sigma cx} y = \mathsf{Some}\, v) \leftrightarrow (\exists n.\ \phi^n_c \langle x, y\rangle = \mathsf{Some}\, v)$$

We can now define the partial application function σ using R and I from Lemma 29.3 and the terms from Fact 29.4

$$\sigma cx := \mathbf{if}\ Rc\ \mathbf{is}\ \mathsf{Some}\, t\ \mathbf{then}\ I(\lambda y.\ t(t_{\mathsf{embed}}(\varepsilon_\mathbb{N} x)y))\ \mathbf{else}\ c$$

Note that $\lambda y.\ t_{\mathsf{embed}}(\varepsilon_\mathbb{N} x)y$ is an L-term, which is translated to a natural number using I.

Theorem 29.9. $\forall cxyv.\ (\exists n.\ \phi^n_{\sigma cx} y = \mathsf{Some}\, v) \leftrightarrow (\exists n.\ \phi^n_c \langle x, y\rangle = \mathsf{Some}\, v)$

Proof. We only prove the direction from right to left, the other direction is similar. Let $\phi^n_c \langle x, y\rangle = \mathsf{Some}\, v$. Then $Rc = \mathsf{Some}\, t$ and $t\ \varepsilon_\mathbb{N}(\langle x, y\rangle) \triangleright v$ for some closed abstraction t. Let $s := (\lambda y.\ t_{\mathsf{embed}}(\varepsilon_\mathbb{N} x)y)\ (\varepsilon_\mathbb{N} y)$. We have $t\ \varepsilon_\mathbb{N}(\langle x, y\rangle) \equiv s$ and thus $s \triangleright v$. Thus there is m s.t. $\mathsf{eval}\, m\, s = \mathsf{Some}\, v$ and we have $\phi^m_{\sigma cx} = \mathsf{Some}\, v$. ∎

Corollary 29.10. $\mathsf{CT}_\mathsf{L} \to \Sigma\phi.\ \mathsf{CT}_\phi \wedge \mathsf{SMN}_\phi$

29.3 Towards mechanised admissibility

CT_L is admissible if for every function $f : \mathbb{N} \to \mathbb{N}$ definable without assumptions (i.e. in an empty context) in CIC, one can externally prove that there exists a proof of $\exists c.\forall x.\exists n.\ \phi^n_c x = \mathsf{Some}(f\, x)$ in CIC. Equivalently, one can ask for a proof of $\exists s : \mathsf{tm}_\mathsf{L}.\forall n : \mathbb{N}.\ s\ \overline{n} \triangleright \overline{f\, n}$ (Theorem 29.5). The present thesis is built on the assumption that CT_L is indeed admissible.

We here outline how a mechanised proof of this fact might look like, based on the verified (type and proof) erasure function from the MetaCoq project [211, 212]. MetaCoq provides types tm_CIC and $\mathsf{ctx}_\mathsf{CIC}$ representing terms and definition contexts of CIC as a deep embedding, an inductive typing predicate $(\_ \vdash_\mathsf{CIC} \_ : \_) : \mathsf{ctx}_\mathsf{CIC} \to \mathsf{tm}_\mathsf{CIC} \to \mathsf{tm}_\mathsf{CIC} \to \mathbb{P}$, and an inductive weak call-by-value evaluation predicate $(\_ \vdash_\mathsf{CIC} \_ \triangleright \_) : \mathsf{ctx}_\mathsf{CIC} \to \mathsf{tm}_\mathsf{CIC} \to \mathsf{tm}_\mathsf{CIC} \to \mathbb{P}$.

The erasure function is a function $\mathsf{erase} : \mathsf{ctx}_\mathsf{CIC} \to \mathsf{tm}_\mathsf{CIC} \to \mathsf{tm}_\square$. The type $\mathsf{tm}_\square$ is similar to the type of terms tm_CIC, but has no constructors to represent types or proofs and instead a

[211] Sozeau et al.. 2020. The MetaCoq Project.

[212] Sozeau et al.. 2019. Coq Coq correct! Verification of type checking and erasure for Coq, in Coq.

constructor $\square$ which indicates that a sub-term was erased. An operational semantics of $\mathsf{tm}_\square$ can be specified via a weak call-by-value evaluation predicate $(\_ \vdash_\square \_ \triangleright \_) : \mathsf{ctx}_\square \to \mathsf{tm}_\square \to \mathsf{tm}_\square \to \mathbb{P}$.

The verified type and proof erasure function preserves evaluation on the base type $\mathbb{N}$. To specify this formally, we require functions $\varepsilon_{\mathsf{CIC}} : \mathbb{N} \to \mathsf{tm}_{\mathsf{CIC}}$ and $\varepsilon_\square : \mathbb{N} \to \mathsf{tm}_\square$ which encode numbers in the respective calculi via application and representations of the constructors S and 0. The correctness theorem of erase relies on an assumption of strong normalisation for CIC.

[212] Sozeau, Boulier, Forster, Tabareau, and Winterhalter. 2019. Coq Coq correct! Verification of type checking and erasure for Coq, in Coq.

Theorem 29.11 ([212, Corollary 4.8.1]). If Σ contains no axioms and $\Sigma \vdash_{\mathsf{CIC}} n : \mathbb{N}$, then one can obtain m such that $\Sigma \vdash_{\mathsf{CIC}} n \triangleright \varepsilon_{\mathsf{CIC}} m$ and $\mathsf{map\ erase}\ \Sigma \vdash_\square \mathsf{erase}\ n \triangleright \varepsilon_\square m$.

Verbalised, the theorem states that every a $n : \mathbb{N}$ evaluates to a value which is just an iterated application of constructors S and 0, and that its erasure evaluates to the same term.

To prove that CT is admissible, we require that the proof of $\mathsf{map\ erase}\ \Sigma \vdash_\square \mathsf{erase}\ n \triangleright \varepsilon_\square m$ can actually be turned into a proof H s.t. $\emptyset \vdash_{\mathsf{CIC}} H : (\mathsf{map\ erase}\ \Sigma \dot{\vdash}_\square \mathsf{erase}\ n \triangleright \varepsilon_\square m)$. By $\dot{\vdash}_\square$ we mean the predicate $\vdash_\square$ turned into a predicate of the $\vdash_{\mathsf{CIC}}$ type system. We note this as a conjecture.

Conjecture 29.12. If Σ contains no axioms and $\Sigma \vdash_{\mathsf{CIC}} n : \mathbb{N}$, then one can obtain m such that $\Sigma \vdash_{\mathsf{CIC}} n \triangleright \varepsilon_{\mathsf{CIC}} m$ and one can obtain $H : \mathsf{tm}_{\mathsf{CIC}}$ such that $\Sigma \vdash_{\mathsf{CIC}} H : (\mathsf{map\ erase}\ \Sigma \dot{\vdash}_\square \mathsf{erase}\ n \triangleright \varepsilon_\square m)$.

The proof of the conjecture likely would be similar to the semantic verification of type and proof erasure by [156], but simpler since we only are interested in axiom-free contexts.

[156] Letouzey. 2004. Programmation fonctionnelle certifiée: l'extraction de programmes dans l'assistant Coq.

By implementing a step-indexed interpreter for the evaluation relation of the $\mathsf{tm}_\square$ calculus and extracting it to L, one could prove the following conjecture. Note that since $\mathsf{tm}_\square$ is a nested inductive type, the extraction requires defining the shape of $\mathsf{tm}_\square$.

Conjecture 29.13. There is an L-term $t_\triangleright$ such that if $\Sigma \vdash_\square s \triangleright v$, then $t_\triangleright(\varepsilon_{\mathsf{tm}_\square} s) \triangleright \varepsilon_{\mathsf{tm}_\square} v$.

The two conjectures together would lead to the admissibility of CT_L.

Conjecture 29.14. If Σ contains no axioms and $\Sigma; \emptyset \vdash_{\mathsf{CIC}} f : \mathbb{N} \to \mathbb{N}$, then one can obtain a proof term $H : \mathsf{tm}_{\mathsf{CIC}}$ such that $\Sigma; \emptyset \vdash_{\mathsf{CIC}} H : (\exists s : \mathsf{tm}_\mathsf{L}. \forall n : \mathbb{N}.\ s\ \overline{n} \triangleright \overline{f\,n})$.

A mechanised consistency proof of CT for CIC might be possible by syntactically analysing the normal form of a proof term H with $\Sigma \vdash_{\mathsf{CIC}} H : \mathsf{CT} \to \bot$ for an axiom-free Σ. This will be a function $\lambda C : \mathsf{CT}.t$, where the term t will have n applications of C to a function f. By applying admissibility of CT to these n instances, one might obtain a proof term H' such that $\Sigma \vdash_{\mathsf{CIC}} H' : \bot$, which contradicts the consistency of $\vdash_{\mathsf{CIC}}$ provable from the assumed strong normalisation. Even more interesting opportunities for future work arise by the questions whether and how Conjectures 29.12 and 29.13 can be generalised to contexts Σ assuming axioms and whether this can give rise to a consistency proof of $\mathsf{LEM} \wedge \mathsf{CT}$.

Bibliography

[1] Wilhelm Ackermann. 1928. Zum Hilbertschen Aufbau der reellen Zahlen. *Math. Ann.* 99, 1 (1928), 118–133. https://doi.org/10.1007/BF01459088

[2] Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, and Matthew Weaver. 2017. CertiCoq: A verified compiler for Coq. In *The third international workshop on Coq for programming languages (CoqPL)*.

[3] Peter B. Andrews. 2002. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Springer Netherlands. https://doi.org/10.1007/978-94-015-9934-4

[4] Andrea Asperti and Wilmer Ricciotti. 2012. Formalizing Turing Machines. In *Logic, Language, Information and Computation*. Springer, 1–25. https://doi.org/10.1007/978-3-642-32621-9_1

[5] Andrea Asperti and Wilmer Ricciotti. 2015. A formalization of multi-tape Turing machines. *Theoretical Computer Science* 603 (2015), 23–42. https://doi.org/10.1016/j.tcs.2015.07.013

[6] Jeremy Avigad, Leonardo de Moura, and Soonho Kong. 2015. Theorem proving in Lean. https://leanprover.github.io/tutorial/tutorial.pdf

[7] Yehoshua Bar-Hillel, Micha Perles, and Eli Shamir. 1961. On formal properties of simple phrase structure grammars. *Sprachtypologie und Universalienforschung* 14 (1961), 143–172.

[8] Henk Barendregt, Wil Dekkers, and Richard Statman. 2013. *Lambda calculus with types*. Cambridge University Press. https://doi.org/10.1017/CBO9781139032636

[9] Bruno Barras. 2010. Sets in Coq, Coq in Sets. *Journal of Formalized Reasoning* 3, 1 (2010), 29–48. https://doi.org/10.6092/issn.1972-5787/1695

[10] Andrej Bauer. 2006. First steps in synthetic computability theory. *Electronic Notes in Theoretical Computer Science* 155 (2006), 5–31. https://doi.org/10.1016/j.entcs.2005.11.049

[11] Andrej Bauer. 2006. König's Lemma and Kleene Tree. *unpublished note* (2006).

[12] Andrej Bauer. 2015. An injection from the Baire space to natural numbers. *Mathematical Structures in Computer Science* 25, 7 (2015), 1484–1489. https://doi.org/10.1017/S0960129513000406

[13] Andrej Bauer. 2017. On fixed-point theorems in synthetic computability. *Tbilisi Mathematical Journal* 10, 3 (2017), 167–181. https://doi.org/10.1515/tmj-2017-0107

[14] Andrej Bauer. 2020. Synthetic mathematics with an excursion into computability theory (slide set). *University of Wisconsin Logic seminar* (2020). http://math.andrej.com/asset/data/madison-synthetic-computability-talk.pdf

[15] Michael J. Beeson. 1985. *Foundations of Constructive Mathematics*. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-68952-9

[16] Christoph Benzmüller and Peter Andrews. 2019. Church's Type Theory. In *The Stanford Encyclopedia of Philosophy* (summer 2019 ed.), Edward N. Zalta (Ed.). Metaphysics Research Lab, Stanford University.

[17] Josef Berger and Hajime Ishihara. 2005. Brouwer's fan theorem and unique existence in constructive analysis. *Mathematical Logic Quarterly* 51, 4 (2005), 360–364. https://doi.org/10.1002/malq.200410038

[18] Josef Berger, Hajime Ishihara, and Peter Schuster. 2012. The weak König lemma, Brouwer's fan theorem, De Morgan's law, and dependent choice. *Reports on Mathematical Logic* 47 (2012), 63. https://doi.org/10.4467/20842589RM.12.003.0684

[19] Robert Berger. 1966. *The undecidability of the domino problem*. Number 66. American Mathematical Society. https://doi.org/10.1090/memo/0066

[20] Yves Bertot and Pierre Castéran. 2013. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media. https://doi.org/10.1007/978-3-662-07964-5

[21] Evert W. Beth. 1948. Semantical Considerations on Intuitionistic Mathematics. *Journal of Symbolic Logic* 13, 3 (1948), 173–173. https://doi.org/10.2307/2267876

[22] Katalin Bimbó. 2015. The decidability of the intensional fragment of classical linear logic. *Theoretical Computer Science* 597 (Sept. 2015), 1–17. https://doi.org/10.1016/j.tcs.2015.06.019

[23] Errett Bishop and Douglas Bridges. 2012. *Constructive analysis*. Vol. 279. Springer Science & Business Media. https://doi.org/10.1017/CBO9780511565663.003

[24] Egon Börger, Erich Grädel, and Yuri Gurevich. 2001. *The classical decision problem*. Springer Science & Business Media. https://doi.org/10.1142/9789812794499_0020

[25] Douglas Bridges and Fred Richman. 1987. *Varieties of constructive mathematics*. Vol. 97. Cambridge University Press. https://doi.org/10.1017/CBO9780511565663

[26] Mario Carneiro. 2019. Formalizing Computability Theory via Partial Recursive Functions. In *10th International Conference on Interactive Theorem Proving (ITP 2019) (Leibniz International Proceedings in Informatics (LIPIcs))*, John Harrison, John O'Leary, and Andrew Tolmach (Eds.), Vol. 141. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 12:1–12:17. https://doi.org/10.4230/LIPIcs.ITP.2019.12

[27] Elliot Catt and Michael Norrish. 2021. On the formalisation of Kolmogorov complexity. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM. https://doi.org/10.1145/3437992.3439921

[28] Kaustuv Chaudhuri. 2018. Expressing additives using multiplicatives and subexponentials. *Mathematical Structures in Computer Science* 28, 5 (2018), 651–666. https://doi.org/10.1017/S0960129516000293

[29] Noam Chomsky. 1956. Three models for the description of language. *IRE Transactions on Information Theory* 2, 3 (1956), 113–124. https://doi.org/10.1109/TIT.1956.1056813

[30] Alonzo Church. 1932. A set of postulates for the foundation of logic. *Annals of mathematics* (1932), 346–366. https://doi.org/10.2307/1968702

[31] Alonzo Church. 1936. An unsolvable problem of elementary number theory. *American journal of mathematics* 58, 2 (1936), 345–363. https://doi.org/10.2307/2371045

[32] Alberto Ciaffaglione. 2016. Towards Turing computability via coinduction. *Science of Computer Programming* 126 (2016), 31–51. https://doi.org/10.1016/j.scico.2016.02.004

[33] Cyril Cohen and Assia Mahboubi. 2010. A Formal Quantifier Elimination for Algebraically Closed Fields. In *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 189–203. https://doi.org/10.1007/978-3-642-14128-7_17

[34] The Coq community. 2021. Awesome Coq: A curated list of awesome Coq libraries, plugins, tools, and resources (GitHub repository). (2021). https://github.com/coq-community/awesome-coq accessed May 7th 2021.

[35] John H Conway. 1987. Fractran: A simple universal programming language for arithmetic. In *Open Problems in Communication and Computation*. Springer, 4–26. https://doi.org/10.1007/978-1-4612-4808-8_2

[36] S Barry Cooper. 2003. *Computability theory*. CRC Press. https://doi.org/10.1201/9781315275789

[37] B. J. Copeland (Ed.). 2004. *The Essential Turing*. Oxford University Press. https://doi.org/10.1093/oso/9780198250791.001.0001

[38] Thierry Coquand. 1989. *Metamathematical investigations of a calculus of constructions*. Technical Report RR-1088. INRIA. https://hal.inria.fr/inria-00075471

[39] Thierry Coquand and Gérard P Huet. 1988. The Calculus of Constructions. *Information and Computation* 76, 2/3 (1988), 95–120. https://doi.org/10.1016/0890-5401(88)90005-3

[40] Thierry Coquand and Bassel Mannaa. 2017. The Independence of Markov's Principle in Type Theory. *Logical Methods in Computer Science ; Volume 13* (2017), Issue 3 ; 1860–5974. https://doi.org/10.23638/LMCS-13(3:10)2017

[41] Luís Cruz-Filipe, Herman Geuvers, and Freek Wiedijk. 2004. C-CoRN, the constructive Coq repository at Nijmegen. In *International Conference on Mathematical Knowledge Management*. Springer, 88–103. https://doi.org/10.1007/978-3-540-27818-4_7

[42] Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. 2021. Formalising a Turing-Complete Choreographic Language in Coq. In *12th International Conference on Interactive Theorem Proving (ITP 2021) (Leibniz International Proceedings in Informatics (LIPIcs))*, Liron Cohen and Cezary Kaliszyk (Eds.), Vol. 193. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 15:1–15:18. https://doi.org/10.4230/LIPIcs.ITP.2021.15

[43] Pierre-Louis Curien and Giorgio Ghelli. 1990. Coherence of subsumption. In *CAAP '90*. Springer Berlin Heidelberg, 132–146. https://doi.org/10.1007/3-540-52590-4_45

[44] Nigel Cutland. 1980. *Computability*. Cambridge University Press. https://doi.org/10.1017/cbo9781139171496

[45] Al Danial. 2021. cloc: Count Lines of Code (GitHub repository). (2021). https://github.com/AlDanial/cloc accessed May 7th 2021.

[46] Martin Davis. 1982. Why Gödel Didn't Have Church's Thesis. *Information and Control* 54, 1/2 (1982), 3–24. https://doi.org/10.1016/S0019-9958(82)91226-8

[47] Martin D. Davis. 1958. *Computability and Unsolvability*. McGraw-Hill.

[48] Martin D. Davis and Elaine J. Weyuker. 1983. *Computability, complexity, and languages - fundamentals of theoretical computer science*. Academic Press.

[49] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. 2015. The Lean theorem prover (system description). In *International Conference on Automated Deduction*. Springer, 378–388. https://doi.org/10.1007/978-3-319-21401-6_26

[50] Richard Dedekind. 1888. *Was sind und was sollen die Zahlen?* (1 ed.). Verlag von Friedrich Vieweg und Sohn, Braunschweig. https://doi.org/10.1007/978-3-663-02788-1 Das Buch wurde unverändert in vielen weiteren Auflagen gedruckt. Die ISBN verweist auf einen aktuellen Nachdruck der zweiten Auflage.

[51] Oliver Deiser. 2009. A Simple Continuous Bijection from Natural Sequences to Dyadic Sequences. *The American Mathematical Monthly* 116, 7 (2009), 643–646. https://doi.org/10.1080/00029890.2009.11920983

[52] James C. E. Dekker. 1954. A theorem on hypersimple sets. *Proc. Amer. Math. Soc.* 5 (1954), 791–796. https://doi.org/10.1090/S0002-9939-1954-0063995-6

[53] The CoRn development team. 2021. CoRN: Coq Repository at Nijmegen. (2021). https://github.com/coq-community/corn accessed May 7th 2021.

[54] Radu Diaconescu. 1975. Axiom of choice and complementation. *Proc. Amer. Math. Soc.* 51, 1 (1975), 176–178. https://doi.org/10.1090/S0002-9939-1975-0373893-X

[55] Hannes Diener. 2020. Constructive Reverse Mathematics. *arXiv:1804.05495 [math]* (2020). arXiv:math/1804.05495 https://arxiv.org/abs/1804.05495

[56] Hannes Diener and Hajime Ishihara. 2021. Bishop-Style Constructive Reverse Mathematics. In *Theory and Applications of Computability*. Springer International Publishing, 347–365. https://doi.org/10.1007/978-3-030-59234-9_10

[57] Christian Doczkal and Joachim Bard. 2018. Completeness and decidability of converse PDL in the constructive type theory of Coq. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM. https://doi.org/10.1145/3167088

[58] Christian Doczkal and Gert Smolka. 2014. Completeness and Decidability Results for CTL in Coq. In *Interactive Theorem Proving*. Springer International Publishing, 226–241. https://doi.org/10.1007/978-3-319-08970-6_15

[59] Gilles Dowek. 1993. The undecidability of typability in the lambda-pi-calculus. In *International Conference on Typed Lambda Calculi and Applications*. Springer, 139–145. https://doi.org/10.1007/BFb0037103

[60] Gilles Dowek. 2001. Higher-Order Unification and Matching. *Handbook of automated reasoning* 2 (2001), 1009–1062. https://doi.org/10.1016/B978-044450813-3/50018-7

[61] Andrej Dudenhefner. 2020. Undecidability of Semi-Unification on a Napkin. In *5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, June 29-July 6, 2020, Paris, France (Virtual Conference) (LIPIcs)*, Zena M. Ariola (Ed.), Vol. 167. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 9:1–9:16. https://doi.org/10.4230/LIPIcs.FSCD.2020.9

[62] Andrej Dudenhefner. 2021. Constructive Many-one Reduction from the Halting Problem to Semi-unification. (2021).

[63] Andrej Dudenhefner. 2021. The Undecidability of System F Typability and Type Checking for Reductionists. (2021).

[64] Andrej Dudenhefner and Jakob Rehof. 2018. A Simpler Undecidability Proof for System F Inhabitation. In *24th International Conference on Types for Proofs and Programs, TYPES 2018, June 18-21, 2018, Braga, Portugal (LIPIcs)*, Peter Dybjer, José Espírito Santo, and Luís Pinto (Eds.), Vol. 130. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2:1–2:11. https://doi.org/10.4230/LIPIcs.TYPES.2018.2

[65] Martín H. Escardó and Cory M. Knapp. 2017. Partial Elements and Recursion via Dominances in Univalent Type Theory. In *26th EACSL Annual Conference on Computer Science Logic (CSL 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Valentin Goranko and Mads Dam (Eds.), Vol. 82. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 21:1–21:16. https://doi.org/10.4230/LIPIcs.CSL.2017.21

[66] Solomon Feferman, John W. Dawson, Stephen C. Kleene, Gregory H. Moore, Robert M. Solovay, and Jean van Heijenoort (Eds.). 1986. *Kurt Godel: Collected Works. Vol. 1: Publications 1929-1936*. Oxford University Press, Inc., USA.

[67] Thiago Mendonça Ferreira Ramos, Ariane Alves Almeida, and Mauricio Ayala-Rincón. 2020. *Formalization of Rice's Theorem over a Functional Language Model*. Technical Report. https://www.mat.unb.br/~ayala/RiceThFormalization.pdf

[68] Denis Firsov and Tarmo Uustalu. 2015. Dependently typed programming with finite sets. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*. 33–44. https://doi.org/10.1145/2808098.2808102

[69] Yannick Forster. 2014. *A Formal and Constructive Theory of Computation*. Bachelor's Thesis. Bachelor's Thesis, Saarland University.

[70] Yannick Forster. 2021. Church's Thesis and Related Axioms in Coq's Type Theory. In *29th EACSL Annual Conference on Computer Science Logic (CSL 2021) (Leibniz International Proceedings in Informatics (LIPIcs))*, Christel Baier and Jean Goubault-Larrecq (Eds.), Vol. 183. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 21:1–21:19. https://doi.org/10.4230/LIPIcs.CSL.2021.21

[71] Yannick Forster, Edith Heiter, and Gert Smolka. 2018. Verification of PCP-related computational reductions in Coq. In *International Conference on Interactive Theorem Proving*. Springer, 253–269. https://doi.org/10.1007/978-3-319-94821-8_15

[72] Yannick Forster, Felix Jahn, and Gert Smolka. 2021. A Constructive and Synthetic Theory of Reducibility: Myhill's Isomorphism Theorem and Post's Problem for Many-one and Truth-table Reducibility in Coq. (2021). https://www.ps.uni-saarland.de/Publications/documents/ForsterEtAl_2020_Synthetic-Reducibility-in-Coq.pdf Pre-print.

[73] Yannick Forster, Dominik Kirst, and Gert Smolka. 2019. On synthetic undecidability in Coq, with an application to the Entscheidungsproblem. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs - CPP 2019*. ACM Press. https://doi.org/10.1145/3293880.3294091

[74] Yannick Forster, Dominik Kirst, and Dominik Wehr. 2020. Completeness Theorems for First-Order Logic Analysed in Constructive Type Theory. In *International Symposium on Logical Foundations of Computer Science*. Springer, 47–74. https://doi.org/10.1007/978-3-030-36755-8_4

[75] Yannick Forster, Dominik Kirst, and Dominik Wehr. 2021. Completeness theorems for first-order logic analysed in constructive type theory. *Journal of Logic and Computation* 31, 1 (Jan. 2021), 112–151. https://doi.org/10.1093/logcom/exaa073

[76] Yannick Forster and Fabian Kunze. 2019. A Certifying Extraction with Time Bounds from Coq to Call-By-Value Lambda Calculus. In *10th International Conference on Interactive Theorem Proving (ITP 2019) (Leibniz International Proceedings in Informatics (LIPIcs))*, John Harrison, John O'Leary, and Andrew Tolmach (Eds.), Vol. 141. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 17:1–17:19. https://doi.org/10.4230/LIPIcs.ITP.2019.17

[77] Yannick Forster, Fabian Kunze, and Marc Roth. 2020. The weak call-by-value λ-calculus is reasonable for both time and space. *Proceedings of the ACM on Programming Languages* 4, POPL (Jan. 2020), 1–23. https://doi.org/10.1145/3371095

[78] Yannick Forster, Fabian Kunze, Gert Smolka, and Maximilian Wuttke. 2021. A Mechanised Proof of the Time Invariance Thesis for the Weak Call-by-value λ-Calculus. 193, 29 (2021).

[79] Yannick Forster, Fabian Kunze, and Maximilian Wuttke. 2020. Verified programming of Turing machines in Coq. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM. https://doi.org/10.1145/3372885.3373816

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

[81] Yannick Forster and Dominique Larchey-Wendling. 2019. Certified undecidability of intuitionistic linear logic via binary stack machines and Minsky machines. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 104–117. https://doi.org/10.1145/3293880.3294096

[82] Yannick Forster, Dominique Larchey-Wendling, Andrej Dudenhefner, Edith Heiter, Dominik Kirst, Fabian Kunze, Gert Smolka, Simon Spies, Dominik Wehr, and Maximilian Wuttke. 2020. A Coq library of undecidable problems. In *The Sixth International Workshop on Coq for Programming Languages (CoqPL 2020)*. https://github.com/uds-psl/coq-library-undecidability

[83] Yannick Forster and Gert Smolka. 2017. Weak Call-by-Value Lambda Calculus as a Model of Computation in Coq. In *Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings (Lecture Notes in Computer Science)*, Mauricio Ayala-Rincón and César A. Muñoz (Eds.), Vol. 10499. Springer, 189–206. https://doi.org/10.1007/978-3-319-66107-0_13

[84] Yannick Forster and Gert Smolka. 2019. Call-by-value lambda calculus as a model of computation in Coq. *Journal of Automated Reasoning* 63, 2 (2019), 393–413. https://doi.org/10.1007/s10817-018-9484-2

[85] R. M. Friedberg. 1957. Two Recursively enumerable sets of incomparable degrees of unsovlability (solution of Post's problem, 1944. *Proceedings of the National Academy of Sciences* 43, 2 (Feb. 1957), 236–238. https://doi.org/10.1073/pnas.43.2.236

[86] Richard M Friedberg and Hartley Rogers Jr. 1959. Reducibility and completeness for sets of integers. *Mathematical Logic Quarterly* 5, 7-13 (1959), 117–125. https://doi.org/10.1002/malq.19590050703

[87] Harvey Friedman. 1978. Classically and intuitionistically provably recursive functions. In *Higher set theory*. Springer, 21–27. https://doi.org/10.1007/BFb0103100

[88] Lennard Gäher and Fabian Kunze. 2021. Mechanising Complexity Theory: The Cook-Levin Theorem in Coq. In *12th International Conference on Interactive Theorem Proving (ITP 2021) (Leibniz International Proceedings in Informatics (LIPIcs))*, Liron Cohen and Cezary Kaliszyk (Eds.), Vol. 193. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 20:1–20:18. https://doi.org/10.4230/LIPIcs.ITP.2021.20

[89] Gerhard Gentzen. 1936. Die Widerspruchsfreiheit der reinen Zahlentheorie. *Math. Ann.* 112, 1 (1936), 493–565. https://doi.org/10.1007/BF01565428

[90] Giorgio Ghelli. 1990. *Proof Theoretic Studies about a minimal type system integrating inclusion and parametric polymorphism*. Ph.D. Dissertation. Università di Pisa. Dipartimento di Informatica.

[91] Giorgio Ghelli. 1995. Divergence of F⩽ type checking. *Theoretical Computer Science* 139, 1-2 (March 1995), 131–162. https://doi.org/10.1016/0304-3975(94)00037-j

[92] Kurt Gödel. 1930. Die Vollständigkeit der Axiome des logischen Funktionenkalküls. *Monatshefte für Mathematik und Physik* 37, 1 (Dec. 1930), 349–360. https://doi.org/10.1007/bf01696781

[93] Kurt Gödel. 1934. *On formally undecidable propositions of Principia Mathematica and related systems*. Lecture at Princeton. https://doi.org/10.1063/1.3051400

[94] Kurt Gödel. 1951. Some basic theorems on the foundations of mathematics and their implications. *Collected Words* (1951), 304–323.

[95] Von Kurt Gödel. 1958. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes. *Dialectica* 12, 3-4 (Dec. 1958), 280–287. https://doi.org/10.1111/j.1746-8361.1958.tb01464.x

[96] Warren D. Goldfarb. 1981. The undecidability of the second-order unification problem. *Theoretical Computer Science* 13 (1981), 225–230. https://doi.org/10.1016/0304-3975(81)90040-2

[97] Warren D. Goldfarb. 1984. The Gödel class with identity is unsolvable. *Bull. Amer. Math. Soc.* 10, 1 (Jan. 1984), 113–116. https://doi.org/10.1090/s0273-0979-1984-15207-8

[98] Warren D. Goldfarb. 1984. The unsolvability of the Gödel class with identity. *Journal of Symbolic Logic* 49, 4 (Dec. 1984), 1237–1252. https://doi.org/10.2307/2274274

[99] Noah Goodman and John Myhill. 1978. Choice Implies Excluded Middle. *Mathematical Logic Quarterly* 24, 25-30 (1978), 461–461. https://doi.org/10.1002/malq.19780242514 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/malq.19780242514

[100] José Grimm, Alban Quadrat, and Carlos Simpson. 2021. Gaia: Implementation of books from Bourbaki's Elements of Mathematics in Coq (GitHub repository). (2021). https://github.com/coq-community/gaia accessed May 7th 2021.

[101] Kurt Gödel. 1932. Ein Spezialfall des Entscheidungsproblems der theoretischen Logik, See [66], 230–234.

[102] VH Hahanyan. 1981. The consistency of some intuitionistic and constructive principles with a set theory. *Studia Logica* 40, 3 (1981), 237–248. https://doi.org/10.1007/BF02584058

[103] Edith Heiter. 2017. *Undecidability of the Post Correspondence Problem in Coq*. Bachelor's Thesis. Saarland University.

[104] Helmut Schwichtenberg. 2005. A Direct Proof of the Equivalence between Brouwer's Fan Theorem and König's Lemma with a Uniqueness Hypothesis. (2005). https://doi.org/10.3217/JUCS-011-12-2086

[105] Matt Hendtlass and Robert Lubarsky. 2016. SEPARATING FRAGMENTS OF WLEM, LPO, and MP. *The Journal of Symbolic Logic* 81, 4 (2016), 1315–1343. https://doi.org/10.1017/jsl.2016.38

[106] Hugo Herbelin. 2010. An Intuitionistic Logic that Proves Markov's Principle. In *2010 25th Annual IEEE Symposium on Logic in Computer Science*. IEEE. https://doi.org/10.1109/lics.2010.49

[107] Jacques Herbrand. 1930. Les bases de la logique Hilbertienne. *Revue de Métaphysique et de Morale* 37, 2 (1930), 243–255.

[108] Jacques Herbrand. 1932. Sur la non-contradiction de l'Arithmétique. *Journal für die reine und angewandte Mathematik* 1932, 166 (1932), 1–8.

[109] Wim H. Hesselink. 2015. Post's Correspondence Problem and the Undecidability of Context-Free Intersection. Manuscript, University of Groningen. http://wimhesselink.nl/pub/whh513.pdf

[110] David Hilbert and Wilhelm Ackermann. 1928. Grundzüge der theoretischen Logik. (1928). https://doi.org/10.1007/978-3-642-65400-8

[111] Ralf Hinze. 2005. Church numerals, twice! (theoretical pearl). *Journal of Functional Programming* 15, 1 (Jan. 2005), 1–13. https://doi.org/10.1017/s0956796804005313

[112] Philip K. Hooper. 1966. The undecidability of the Turing machine immortality problem. *The Journal of Symbolic Logic* 31, 2 (1966), 219–234. https://doi.org/10.2307/2269811

[113] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2006. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA.

[114] Martín Hötzel Escardó and Chuangjie Xu. 2015. The Inconsistency of a Brouwerian Continuity Principle with the Curry–Howard Interpretation. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany. https://doi.org/10.4230/LIPICS.TLCA.2015.153

[115] Jason Z.S. Hu and Ondřej Lhoták. 2019. Undecidability of $D_{<:}$ and its decidable fragments. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–30. https://doi.org/10.1145/3371077

[116] Gérard Pierre Huet. 1972. *Constrained Resolution: A Complete Method for Higher-Order Logic.* Ph.D. Dissertation. Case Western Reserve University, USA. AAI7306307.

[117] Gérard P Huet. 1973. The undecidability of unification in third order logic. *Information and control* 22, 3 (1973), 257–267. https://doi.org/10.1016/S0019-9958(73)90301-X

[118] Lars Hupel and Tobias Nipkow. 2018. A Verified Compiler from Isabelle/HOL to CakeML. In *Programming Languages and Systems*. Springer International Publishing, 999–1026. https://doi.org/10.1007/978-3-319-89884-1_35

[119] J. Martin E. Hyland. 1982. The Effective Topos. In *The L. E. J. Brouwer Centenary Symposium, Proceedings of the Conference held in Noordwijkerhout*. Elsevier, 165–216. https://doi.org/10.1016/s0049-237x(09)70129-6

[120] J. Martin E. Hyland and Andrew M. Pitts. 1989. The Theory of Constructions: Categorical Semantics and Topos-theoretic Models. In *Categories in Computer Science and Logic (Contemporary Mathematics)*, J. W. Gray and A. Scedrov (Eds.), Vol. 92. Amer. Math. Soc, Providence RI, 137–199.

[121] Hajime Ishihara. 2006. Reverse Mathematics in Bishop's Constructive Mathematics. *Philosophia Scientae* CS 6 (Sept. 2006), 43–59. https://doi.org/10.4000/philosophiascientiae.406

[122] Hajime Ishihara. 2006. Weak König's Lemma Implies Brouwer's Fan Theorem: A Direct Proof. *Notre Dame Journal of Formal Logic* 47, 2 (April 2006). https://doi.org/10.1305/ndjfl/1153858649

[123] Hajime Ishihara, Maria Emilia Maietti, Samuele Maschio, and Thomas Streicher. 2018. Consistency of the intensional level of the Minimalist Foundation with Church's thesis and axiom of choice. *Archive for Mathematical Logic* 57, 7-8 (Jan. 2018), 873–888. https://doi.org/10.1007/s00153-018-0612-9

[124] Felix Jahn. 2020. *Synthetic One-One, Many-One, and Truth-Table Reducibility in Coq*. Bachelor's Thesis. Saarland University.

[125] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. 1988. A proper extension of ML with an effective type-assignment. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '88*. ACM Press. https://doi.org/10.1145/73560.73565

[126] Assaf J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. 1993. The undecidability of the semi-unification problem. *Information and Computation* 102, 1 (1993), 83–101. https://doi.org/10.1006/inco.1993.1003

[127] Dominik Kirst and Marc Hermes. 2021. Synthetic Undecidability and Incompleteness of First-Order Axiom Systems in Coq. In *12th International Conference on Interactive Theorem Proving (ITP 2021) (Leibniz International Proceedings in Informatics (LIPIcs))*, Liron Cohen and Cezary Kaliszyk (Eds.), Vol. 193. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 23:1–23:20. https://doi.org/10.4230/LIPIcs.ITP.2021.23

[128] Dominik Kirst and Marc Hermes. 2021. Synthetic Undecidability and Incompleteness of First-Order Axiom Systems in Coq. In *12th International Conference on Interactive Theorem Proving (ITP 2021). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik*.

[129] Dominik Kirst and Dominique Larchey-Wendling. 2020. Trakhtenbrot's Theorem in Coq. In *Automated Reasoning*. Springer International Publishing, 79–96. https://doi.org/10.1007/978-3-030-51054-1_5

[130] Dominik Kirst and Dominique Larchey-Wendling. 2021. Trakhtenbrot's Theorem in Coq: Finite Model Theory through the Constructive Lens. arXiv:cs.LO/2104.14445 https://arxiv.org/abs/2104.14445

[131] Stephen C. Kleene. 1936. λ-definability and recursiveness. *Duke mathematical journal* 2, 2 (1936), 340–353. https://doi.org/10.1215/S0012-7094-36-00227-2

[132] Stephen C. Kleene. 1943. Recursive predicates and quantifiers. *Trans. Amer. Math. Soc.* 53, 1 (1943), 41–73. https://doi.org/10.1090/S0002-9947-1943-0007371-8

[133] Stephen C. Kleene. 1945. On the interpretation of intuitionistic number theory. *The journal of symbolic logic* 10, 4 (1945), 109–124. https://doi.org/10.2307/2269016

[134] Stephen C. Kleene. 1952. *Introduction to metamathematics*. Vol. 483. van Nostrand New York.

[135] Stephen C. Kleene. 1953. Recursive functions and intuitionistic mathematics. *Proceedings of the International Congress of Mathematicians, Cambridge, Mass.* (1953), 679–685.

[136] Steven C. Kleene and Emil L. Post. 1954. The Upper Semi-Lattice of Degrees of Recursive Unsolvability. *The Annals of Mathematics* 59, 3 (May 1954), 379. https://doi.org/10.2307/1969708

[137] Stephen C. Kleene and J. Barkley Rosser. 1935. The inconsistency of certain formal logics. *Annals of Mathematics* (1935), 630–636. https://doi.org/10.2307/1968646

[138] Andrei N Kolmogorov. 1963. On tables of random numbers. *Sankhyā: The Indian Journal of Statistics, Series A* (1963), 369–376. https://doi.org/10.1016/S0304-3975(98)00075-9

[139] Andrei N. Kolmogorov. 1965. Three approaches to the quantitative definition of information. *Problems of information transmission* 1, 1 (1965), 3–11.

[140] Nils Köpp. 2018. *Automatically verified program extraction from proofs with applications to constructive analysis*. Master's thesis. LMU Munich. http://www.mathematik.uni-muenchen.de/~schwicht/seminars/semws18/main.pdf

[141] Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017. The essence of higher-order concurrent separation logic. In *European Symposium on Programming*. Springer, 696–723. https://doi.org/10.1007/978-3-662-54434-1_26

[142] Georg Kreisel. 1965. Mathematical logic. *Lectures in modern mathematics* 3 (1965), 95–195. https://doi.org/10.2307/2315573

[143] Georg Kreisel. 1970. Church's thesis: a kind of reducibility axiom for constructive mathematics. In *Studies in Logic and the Foundations of Mathematics*. Vol. 60. 121–150. https://doi.org/10.1016/S0049-237X(08)70746-8

[144] Antonín Kučera. [n. d.]. An alternative, priority-free, solution to Post's problem. In *Lecture Notes in Computer Science*. Springer-Verlag, 493–500. https://doi.org/10.1007/bfb0016275

[145] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML. *ACM SIGPLAN Notices* 49, 1 (Jan. 2014), 179–191. https://doi.org/10.1145/2578855.2535841

[146] Martin Kummer. 1996. On the complexity of random strings. In *Annual Symposium on Theoretical Aspects of Computer Science*. Springer, 25–36. https://doi.org/10.1007/3-540-60922-9_3

[147] Fabian Kunze, Gert Smolka, and Yannick Forster. 2018. Formal small-step verification of a call-by-value lambda calculus machine. In *Asian Symposium on Programming Languages and Systems*. Springer, 264–283. https://doi.org/10.1007/978-3-030-02768-1_15

[148] Ugo Dal Lago and Beniamino Accattoli. 2017. Encoding Turing Machines into the Deterministic Lambda-Calculus. *arXiv preprint arXiv:1711.10078* (2017). https://arxiv.org/abs/1711.10078

[149] Ugo Dal Lago and Simone Martini. 2008. The weak lambda calculus as a reasonable machine. *Theoretical Computer Science* 398, 1-3 (2008), 32–50. https://doi.org/10.1016/j.tcs.2008.01.044

[150] Dominique Larchey-Wendling. 2017. Typing total recursive functions in Coq. In *International Conference on Interactive Theorem Proving*. Springer, 371–388. https://doi.org/10.1007/978-3-319-66107-0_24

[151] Dominique Larchey-Wendling. 2021. Synthetic Undecidability of MSELL via FRACTRAN mechanised in Coq. (2021).

[152] Dominique Larchey-Wendling and Yannick Forster. 2019. Hilbert's Tenth Problem in Coq. (2019). https://doi.org/10.4230/LIPICS.FSCD.2019.27

[153] Dominique Larchey-Wendling and Yannick Forster. 2020. Hilbert's Tenth Problem in Coq. *arXiv preprint arXiv:2003.04604* (2020). https://arxiv.org/abs/2003.04604

[154] Dominique Larchey-Wendling and Didier Galmiche. 2010. The undecidability of boolean BI through phase semantics. In *2010 25th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 140–149. https://doi.org/10.1109/LICS.2010.18

[155] Dominique Larchey-Wendling and Jean-François Monin. 2021. The Braga Method: Extracting Certified Algorithms from Complex Recursive Schemes in Coq. In *Proof and Computation: From Proof Theory and Univalent Mathematics to Program Extraction and Verification*, Klaus Mainzer, Peter Schuster, and Helmut Schwichtenberg (Eds.). World Scientific Singapore.

[156] Pierre Letouzey. 2004. *Programmation fonctionnelle certifiée: l'extraction de programmes dans l'assistant Coq*. Ph.D. Dissertation. http://www.pps.jussieu.fr/~letouzey/download/these_letouzey.pdf

[157] Patrick Lincoln, John Mitchell, Andre Scedrov, and Natarajan Shankar. 1992. Decision problems for propositional linear logic. *Annals of pure and Applied Logic* 56, 1-3 (1992), 239–311. https://doi.org/10.1016/0168-0072(92)90075-B

[158] Martin H. Löb. 1976. Embedding first order predicate logic in fragments of intuitionistic logic. *Journal of Symbolic Logic* 41, 4 (Dec. 1976), 705–718. https://doi.org/10.2307/2272390

[159] Salvador Lucas. 2021. The origins of the halting problem. *Journal of Logical and Algebraic Methods in Programming* 121 (June 2021), 100687. https://doi.org/10.1016/j.jlamp.2021.100687

[160] Claudio L. Lucchesi. 1972. The undecidability of the unification problem for third order languages. *Report CSRR* 2059 (1972), 129–198.

[161] Zhaohui Luo. 1994. *Computation and Reasoning: A Type Theory for Computer Science*. Oxford University Press, Inc., USA.

[162] Assia Mahboubi and Enrico Tassi. 2021. *Mathematical Components*. Zenodo. https://doi.org/10.5281/zenodo.4457887

[163] Petar Maksimović and Alan Schmitt. 2015. HOCore in Coq. In *Interactive Theorem Proving*. Springer International Publishing, 278–293. https://doi.org/10.1007/978-3-319-22102-1_19

[164] Zohar Manna. 2003. *Mathematical Theory of Computation*. Dover Publications, Inc., USA.

[165] Andrei A. Markov. 1951. Impossibility of Certain Algorithms in the Theory of Associative Systems. *Journal of Symbolic Logic* 16, 3 (1951), 215–215. https://doi.org/10.2307/2266407

[166] Andrei A. Markov. 1954. The theory of algorithms. *Trudy Matematicheskogo Instituta Imeni VA Steklova* 42 (1954), 3–375. https://doi.org/10.1007/978-94-017-3477-6

[167] The mathlib Community. 2020. The Lean Mathematical Library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2020)*. Association for Computing Machinery, New York, NY, USA, 367–381. https://doi.org/10.1145/3372885.3373824

[168] Yuri V. Matijasevič. 1970. Enumerable sets are Diophantine. In *Soviet Mathematics: Doklady*, Vol. 11. 354–357.

[169] David Charles McCarty. 1991. Incompleteness in intuitionistic metamathematics. *Notre Dame journal of formal logic* 32, 3 (1991), 323–358. https://doi.org/10.1305/ndjfl/1093635833

[170] Marvin L. Minsky. 1967. *Computation: Finite and Infinite Machines*. Prentice-Hall, Inc., USA.

[171] Torben Æ. Mogensen. 1994. Efficient Self-Interpretation in Lambda Calculus. *Journal of Functional Programming* 2 (1994), 345–364. https://doi.org/10.1017/S0956796800000423

[172] Albert Abramovich Muchnik. 1963. On strong and weak reducibility of algorithmic problems. *Sibirskii Matematicheskii Zhurnal* 4, 6 (1963), 1328–1341.

[173] Eric Mullen, Stuart Pernsteiner, James R. Wilcox, Zachary Tatlock, and Dan Grossman. 2018. Œuf: minimizing the Coq extraction TCB. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM. https://doi.org/10.1145/3167089

[174] John Myhill. 1957. Creative sets. (1957). https://doi.org/10.1002/malq.19550010205

[175] Magnos O. Myreen and Scott Owens. 2014. Proof-producing translation of higher-order logic into pure and stateful ML. *Journal of Functional Programming* 24, 2-3 (2014), 284–315. https://doi.org/10.1017/S0956796813000282

[176] Nikolai M. Nagorny. 1995. Andrei markov and mathematical constructivism. In *Studies in Logic and the Foundations of Mathematics*. Vol. 134. Elsevier, 467–479. https://doi.org/10.1016/S0049-237X(06)80057-1

[177] Ulf Norell. 2008. Dependently typed programming in Agda. In *International school on advanced functional programming*. Springer, 230–266. https://doi.org/10.1007/978-3-642-04652-0_5

[178] Michael Norrish. 2011. Mechanised Computability Theory. In *ITP 2011 (LNCS)*, Vol. 6898. Springer, 297–311. https://doi.org/10.1007/978-3-642-22863-6_22

[179] Per Martin-Löf (notes by Giovanni Sambi). 1984. *Intuitionistic type theory*. Vol. 9. Bibliopolis Naples.

[180] Piergiorgio Odifreddi. 1992. *Classical recursion theory: The theory of functions and sets of natural numbers*. Elsevier.

[181] Christine Paulin-Mohring. 1993. Inductive definitions in the system Coq rules and properties. In *International Conference on Typed Lambda Calculi and Applications*. Springer, 328–345. https://doi.org/10.1007/BFb0037116

[182] Christine Paulin-Mohring. 2015. Introduction to the Calculus of Inductive Constructions. https://hal.inria.fr/hal-01094195

[183] Pierre-Marie Pédrot. 2015. *A Materialist Dialectica*. Theses. Paris Diderot. https://hal.archives-ouvertes.fr/tel-01247085

[184] Pierre-Marie Pédrot and Nicolas Tabareau. 2018. Failure is Not an Option. In *European Symposium on Programming*. Springer, 245–271. https://doi.org/10.1007/978-3-319-89884-1_9

[185] Benjamin C. Pierce. 1994. Bounded quantification is undecidable. *Information and Computation* 112, 1 (1994), 131–165. https://doi.org/10.1006/inco.1994.1055

[186] Gordon D. Plotkin. 1975. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science* 1, 2 (Dec. 1975), 125–159. https://doi.org/10.1016/0304-3975(75)90017-1

[187] Emil L. Post. 1936. Finite combinatory processes—formulation. *Journal of Symbolic Logic* 1, 3 (Sept. 1936), 103–105. https://doi.org/10.2307/2269031

[188] Emil L. Post. 1943. Formal reductions of the general combinatorial decision problem. *American journal of mathematics* 65, 2 (1943), 197–215. https://doi.org/10.2307/2371809

[189] Emil L. Post. 1944. Recursively enumerable sets of positive integers and their decision problems. *bulletin of the American Mathematical Society* 50, 5 (1944), 284–316. https://doi.org/10.1090/S0002-9904-1944-08111-1

[190] Emil L. Post. 1946. A variant of a recursively unsolvable problem. *Bull. Amer. Math. Soc.* 52, 4 (1946), 264–268. https://doi.org/10.1090/S0002-9904-1946-08555-9

[191] Emil L. Post. 1947. Recursive unsolvability of a problem of Thue. *The Journal of Symbolic Logic* 12, 1 (1947), 1–11. https://doi.org/10.2307/2267170

[192] Emil L. Post. 1994. Absolutely unsolvable problems and relatively undecidable propositions–Account of an anticipation (1941). *Collected Works of Post* (1994), 375–441.

[193] Damien Pous. 2004. *A certified compiler from recursive functions to Minski machines*. Technical Report. Université Paris-Diderot, PPS. http://perso.ens-lyon.fr/damien.pous/recursive-minski/

[194] James F. Power. 2013. Thue's 1914 paper: a translation. *arXiv preprint arXiv:1308.5858* (2013). https://arxiv.org/abs/1308.5858

[195] Pierre Pradic and Chad E. Brown. 2019. Cantor-Bernstein implies Excluded Middle. *CoRR* abs/1904.09193 (2019). arXiv:1904.09193 http://arxiv.org/abs/1904.09193

[196] Thiago Mendonça Ferreira Ramos, César Muñoz, Mauricio Ayala-Rincón, Mariano Moscato, Aaron Dutle, and Anthony Narkawicz. 2018. Formalization of the Undecidability of the Halting Problem for a Functional Language. In *International Workshop on Logic, Language, Information, and Computation*. Springer, 196–209. https://doi.org/10.1007/978-3-662-57669-4_11

[197] Henry G. Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.* 74, 2 (1953), 358–366. https://doi.org/10.1090/S0002-9947-1953-0053041-6

[198] Fred Richman. 1983. Church's thesis without tears. *The Journal of symbolic logic* 48, 3 (1983), 797–803. https://doi.org/10.2307/2273473

[199] Fred Richman. 2000. The fundamental theorem of algebra: a constructive development without choice. *Pacific J. Math.* 196, 1 (2000), 213–230. https://doi.org/10.2140/pjm.2000.196.213

[200] Fred Richman. 2001. Constructive Mathematics without Choice. In *Reuniting the Antipodes — Constructive and Nonstandard Views of the Continuum*, Peter Schuster, Ulrich Berger, and Horst Osswald (Eds.). Springer Netherlands, Dordrecht, 199–205. https://doi.org/10.1007/978-94-015-9757-9_17

[201] John A. Robinson. 1965. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)* 12, 1 (1965), 23–41. https://doi.org/10.1145/321250.321253

[202] Hartley Rogers. 1987. Theory of Recursive Functions and Effective Computability. (1987).

[203] Steven Schäfer, Gert Smolka, and Tobias Tebbi. 2015. Completeness and decidability of de Bruijn substitution algebra in Coq. In *Proceedings of the 2015 Conference on Certified Programs and Proofs*. 67–73. https://doi.org/10.1145/2676724.2693163

[204] Dana Scott. 1968. A system of functional abstraction. (1968). Lectures delivered at University of California, Berkeley, Cal., 1962/63. Photocopy of a preliminary version, issued by Stanford University, September 1963, furnished by author in 1968.

[205] John C. Shepherdson and H. E. Sturgis. 1963. Computability of Recursive Functions. *Journal of the ACM* 10, 2 (April 1963), 217–255. https://doi.org/10.1145/321160.321170

[206] Yannick Forster Sigurd Schneider, Fabian Kunze. 2021. Smpl: A Coq plugin for forward reasoning (GitHub Repository). (2021). https://github.com/uds-psl/smpl accessed July 14th 2021.

[207] Michael Sipser. 2006. *Introduction to the Theory of Computation*. Vol. 2. Thomson Course Technology Boston. https://doi.org/10.1145/230514.571645

[208] Cees F. Slot and Peter van Emde Boas. 1984. On Tape Versus Core; An Application of Space Efficient Perfect Hash Functions to the Invariance of Space. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1984, Washington, DC, USA*. 391–400. https://doi.org/10.1145/800057.808705

[209] Raymond M. Smullyan. 1994. *Diagonalization and self-reference*. Clarendon Press, Oxford, England.

[210] Robert I. Soare. 1999. *Recursively enumerable sets and degrees: A study of computable functions and computably generated sets*. Springer Science & Business Media.

[211] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. 2020. The MetaCoq Project. *Journal of Automated Reasoning* (2020). https://doi.org/10.1007/s10817-019-09540-0

[212] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. 2019. Coq Coq correct! Verification of type checking and erasure for Coq, in Coq. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–28. https://doi.org/10.1145/3371076

[213] Matthieu Sozeau and Cyprien Mangin. 2019. Equations reloaded: High-level dependently-typed functional programming and proving in Coq. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 1–29. https://doi.org/10.1145/3341690

[214] Sozeau, Matthieu; Harvard University. 2009. A New Look at Generalized Rewriting in Type Theory. *Journal of Formalized Reasoning; Vol 2* (2009), No 1 (2009); 41–62. https://doi.org/10.6092/ISSN.1972-5787/1574

[215] Simon Spies. 2019. *Formalising the Undecidability of Higher-Order Unification*. Bachelor's Thesis. Bachelor's Thesis, Saarland University.

[216] Simon Spies and Yannick Forster. 2020. Undecidability of higher-order unification formalised in Coq. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 143–157. https://doi.org/10.1145/3372885.3373832

[217] Kathrin Stark. 2019. Mechanising syntax with binders in Coq. https://doi.org/10.22028/D291-30298

[218] Kathrin Stark, Steven Schäfer, and Jonas Kaiser. 2019. Autosubst 2: reasoning with multi-sorted de Bruijn terms and vector substitutions. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs - CPP 2019*. ACM Press. https://doi.org/10.1145/3293880.3294101

[219] Lutz Straßburger. 2019. On the decision problem for MELL. *Theoretical Computer Science* 768 (May 2019), 91–98. https://doi.org/10.1016/j.tcs.2019.02.022

[220] Andrew Swan and Taichi Uemura. 2019. On Church's Thesis in Cubical Assemblies. *arXiv preprint arXiv:1905.03014* (2019). https://arxiv.org/abs/1905.03014

[221] The GeoCoq Development Team. 2021. GeoCoq: A formalization of geometry in Coq. (2021). https://github.com/GeoCoq/GeoCoq accessed May 7th 2021.

[222] The Coq Development Team. 2021. The Coq Proof Assistant version 8.13.2. https://doi.org/10.5281/zenodo.4501022

[223] The Coq Development Team. 2021. The Coq Reference Manual. (2021). https://doi.org/10.5281/zenodo.4021912

[224] The Coq std++ Team. 2020. An extended "standard library" for Coq. https://gitlab.mpi-sws.org/iris/stdpp.

[225] Axel Thue. 1914. Probleme uber Veränderungen von Zeichenreihen nach gegebenen Regeln. *Skrifter utgit av Videnskapsselskapet i Kristiania, I Mathematische und Naturwissenschaftliche Klasse* 10 (1914).

[226] Amin Timany and Matthieu Sozeau. 2018. Cumulative Inductive Types In Coq. 108 (2018), 29:1–29:16. https://doi.org/10.4230/LIPIcs.FSCD.2018.29

[227] Boris A. Trakhtenbrot. 1950. The impossibility of an algorithm for the decidability problem on finite classes. *Doklady Akademii Nauk SSSR* 70, 4 (1950), 569–572.

[228] Anne Sjerp Troelstra and Dirk van Dalen. 1988. Constructivism in mathematics. Vol. I. *Studies in Logic and the Foundations of Mathematics* 26 (1988).

[229] Alan Mathison Turing. 1936. On computable numbers, with an application to the Entscheidungsproblem. *Journal of Math* 58, 345-363 (1936), 5.

[230] Alan Mathison Turing. 1939. Systems of logic based on ordinals. *Proceedings of the London mathematical society* 2, 1 (1939), 161–228. https://doi.org/10.1112/plms/s2-45.1.161

[231] The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study.

[232] Vladimir Voevodsky, Benedikt Ahrens, Daniel Grayson, et al. [n. d.]. UniMath — a computer-checked library of univalent mathematics. available at https://github.com/UniMath/UniMath. https://github.com/UniMath/UniMath

[233] Dominik Wehr. 2019. *A Constructive Analysis of First-Order Completeness Theorems in Coq*. Bachelor's Thesis. Saarland University.

[234] Joe B Wells. 1999. Typability and type checking in System F are equivalent and undecidable. *Annals of Pure and Applied Logic* 98, 1-3 (1999), 111–156. https://doi.org/10.1016/S0168-0072(98)00047-5

[235] Benjamin Werner. 1997. Sets in types, types in sets. In *International Symposium on Theoretical Aspects of Computer Software*. Springer, 530–546. https://doi.org/10.1007/BFb0014566

[236] Wikipedia contributors. 2021. Rice's theorem — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Rice's_theorem&oldid=1017713534. [Online; accessed 31-May-2021].

[237] Wikipedia contributors. 2021. Turing machine — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Turing_machine&oldid=1030525471. [Online; accessed 16-July-2021].

[238] Maximilian Wuttke. 2018. *Verified programming of Turing machines in Coq*. Bachelor's thesis. Saarland University.

[239] Jian Xu, Xingyuan Zhang, and Christian Urban. 2013. Mechanising Turing machines and computability theory in Isabelle/HOL. In *International Conference on Interactive Theorem Proving*. Springer, 147–162. https://doi.org/10.1007/978-3-642-39634-2_13

[240] Norihiro Yamada. 2020. Game semantics of Martin-Löf type theory, part III: its consistency with Church's thesis. arXiv:math.LO/2007.08094 https://arxiv.org/abs/2007.08094

[241] Vincent Zammit. 1997. *A Proof of the S-m-n theorem in Coq*. Technical Report 9-97. The Computing Laboratory, The University of Kent, Canterbury, Kent, UK. 182–196 pages. http://www.cs.kent.ac.uk/pubs/1997/227

APPENDIX A

Basic Definitions and Notation

We here give an overview of inductive types and notations well-known in type theory and constructive mathematics. This chapter is thus mostly to keep the thesis self-contained and not meant for linear reading.

A.1 Inductive types

The inductive types of interest for this paper are **natural numbers**, the **unit type**, **booleans**, **options**, **lists**, **sums**, **pairs**, and **dependent pairs**:

Def. natural numbers
Def. unit type
Def. booleans
Def. options
Def. lists
Def. sums
Def. pairs
Def. dependent pairs

$$n : \mathbb{N} ::= 0 \mid \mathsf{S}\, n \qquad \text{(natural numbers)}$$

$$\mathbb{1} ::= \star \qquad \text{(unit type)}$$

$$b : \mathbb{B} ::= \mathsf{false} \mid \mathsf{true} \qquad \text{(booleans)}$$

$$o : \mathbb{O}A ::= \mathsf{None} \mid \mathsf{Some}\, a \quad \textit{where } a : A \qquad \text{(options)}$$

$$l : \mathbb{L}A ::= [\,] \mid a :: l \quad \textit{where } a : A \qquad \text{(lists)}$$

$$A + B := \mathsf{inl}\, a \mid \mathsf{inr}\, b \quad \textit{where } a : A \textit{ and } b : B \qquad \text{(sums)}$$

$$A \times B := (a, b) \quad \textit{where } a : A \textit{ and } b : B \qquad \text{(pairs)}$$

$$\Sigma x : X.\, Ax := (x_0, a) \quad \textit{where } A : X \to \mathbb{T},\ x_0 : X, \textit{ and } a : Ax_0 \qquad \text{(dependent pairs)}$$

We define the projection functions $\pi_1(x, y) := x$ and $\pi_2(x, y)$, overloading the notation for both pairs of type $A \times B$ and dependent pairs of type $\Sigma x.\, Ax$.

One can easily construct a pairing function $\langle \_, \_ \rangle : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ and for all $f : \mathbb{N} \to \mathbb{N} \to X$ an inverse construction $\lambda\langle n, m\rangle.\, f\, n\, m$ of type $\mathbb{N} \to X$ such that $(\lambda\langle n, m\rangle.\, f\, n\, m)\langle n, m\rangle = f\, n\, m$.

We write $n =_{\mathbb{B}} m$ for the boolean equality decider on $\mathbb{N}$, and $\neg_{\mathbb{B}}$, $\wedge_{\mathbb{B}}$, $\vee_{\mathbb{B}}$ for boolean negation, conjunction, and disjunction.

We oftentimes use the notation **if** o **is** $\mathsf{Some}\, x$ **then** $\ldots$ **else** $\ldots$ for an inline case analysis on options, and similarly for other types like lists.

As is common, we write $[x_1, \ldots, x_n]$ for $x_1 :: \cdots :: x_n :: [\,]$. We write $x \in l$ if x occurs in l, and $l_1 \subseteq l_2$ if $\forall x \in l_1.\, x \in l_2$.

Def. list prefix

We say that l_1 is a **list prefix** of l_2 if $l_1 \sqsubseteq l_2 := \exists l.\, l_1 = l_2 +\!\!+ l$.

We use the functions $\mathsf{map} : (X \to Y) \to \mathbb{L}X \to \mathbb{L}Y$, $\_ +\!\!+ \_ : \mathbb{L}X \to \mathbb{L}X \to \mathbb{L}X$, $\mathsf{filter} : (X \to \mathbb{B}) \to \mathbb{L}X \to \mathbb{L}X$, $\_ \times \_ : \mathbb{L}X \to \mathbb{L}Y \to \mathbb{L}(X \times Y)$, $\_[\_] : \mathbb{L}A \to \mathbb{N} \to \mathbb{O}A$, $\_ \in^{f}_{\mathbb{B}} \_ : X \to \mathbb{L}X \to \mathbb{B}$ for $f : X \to X \to \mathbb{B}$ defined as follows:

$$\mathsf{map}\ f\ [\,] := [\,] \qquad \mathsf{map}\ f\ (x :: l) := f\, x :: \mathsf{map}\ f\ l$$

$$
\begin{array}{rl}
[] +\!\!+\, l_2 := l_2 & (x :: l_1) +\!\!+\, l_2 := x :: (l_1 +\!\!+\, l_2) \\
\mathsf{filter}\ f\ [] := [] & \mathsf{filter}\ f\ (x :: l) := \textbf{if}\ f\,x\ \textbf{then}\ x :: \mathsf{filter}\ f\ l\ \textbf{else}\ \mathsf{filter}\ f\ l \\
[] \times l_2 := [] & (x :: l_1) \times l_2 := \mathsf{map}(\lambda y.\,(x,y))l_2 +\!\!+\, (l_1 \times l_2) \\
[]\,[n] := \mathsf{None} & (x :: l)[n] := \textbf{if}\ n\ \textbf{is}\ \mathsf{S}\,n\ \textbf{then}\ l[n]\ \textbf{else}\ \mathsf{Some}\,x \\
\mathsf{pos}\,f\ x\,l := \mathsf{None} & \mathsf{pos}\,f\ x\,(y :: l) := \textbf{if}\ f\,x\,y\ \textbf{then}\ \mathsf{Some}\,0 \\
 & \qquad \textbf{else if}\ \mathsf{pos}\,f\ x\,l\ \textbf{is}\ \mathsf{Some}\,n\ \textbf{then}\ \mathsf{Some}\,(\mathsf{S}\,n) \\
 & \qquad \textbf{else}\ \mathsf{None} \\
x \in^f_{\mathbb{B}} [] := \mathsf{false} & x \in^f_{\mathbb{B}} (y :: l) := \textbf{if}\ f\,x\,y\ \textbf{then}\ \mathsf{true}\ \textbf{else}\ x \in^f_{\mathbb{B}} l
\end{array}
$$

We define the predicate $\mathsf{Forall}_2 : (X \to Y \to \mathbb{P}) \mathbb{L}X \to \mathbb{L}Y \to \mathbb{P}$ as

$$\frac{}{\mathsf{Forall}_2\ p\ []\ []} \qquad \frac{p\,x\,y \qquad \mathsf{Forall}_2\ p\ l_1\ l_2}{\mathsf{Forall}_2\ p\ (x :: l_1)\ (y :: l_2)}$$

If $n < |l|$ we sometimes abusively assume $l[n]{:}A$. When convenient we use the notation $[f\,x \mid x \in l]$ for $\mathsf{map}\ f\ l$ and the notation $[f\,x \mid x \in l, g\,x]$ for $\mathsf{filter}\ g\ (\mathsf{map}\ f\ l)$.

Finally, we introduce vectors over X as type X^n, where $n{:}\,\mathbb{N}$ with the constructors

$$\frac{}{\mathsf{nil} : X^0} \qquad \frac{v{:}X^n \qquad x{:}X}{x :: v : X^{\mathsf{S}\,n}}$$

We write elements of X^n as $(x_1, \ldots, x_n)$, and otherwise reuse notation for lists for vectors. When convenient we identify X^1 with X on paper.

A.2 Propositions

The universe of propositions $\mathbb{P}$ is impredicative, so e.g. $(\forall X{:}\,\mathbb{P}.\,X){:}\,\mathbb{P}$, and a sub-universe of $\mathbb{T}$, i.e. whenever $P{:}\,\mathbb{P}$ we also have $P{:}\,\mathbb{T}$. Propositions can be combined using the usual connectives $\wedge$, $\vee$, and $\neg$. We denote the true proposition by $\top$ and the false proposition by $\bot$.

We write $\forall x{:}X.\,Ax$ for both dependent functions and logical universal quantification, and $\exists x{:}X.\,Ax$ where $A{:}X \to \mathbb{P}$ for existential quantification. We write $\exists!x{:}X.\,Ax$ to denote that there is a unique x satisfying A, i.e. $\exists x{:}X.\,Ax \wedge \forall y.\,Ay \to x = y$.

A.3 Functions

Besides intensional equality ($=$), we consider other more extensional equivalence relations in this thesis. Amongst them are extensional equality of functions f, g ($\forall x.\,f\,x = g\,x$), extensional equivalence of predicates p, q ($\forall x.\,p\,x \leftrightarrow q\,x$), or range equivalence of functions f, g ($\forall x.\,(\exists y.\,f\,y = x) \leftrightarrow (\exists y.\,g\,y = x)$). We denote all of these equivalence relations with the symbol $\equiv$ and indicate what is meant by an index. For discrete X (e.g. $\mathbb{N}$, $\mathbb{ON}$, $\mathbb{LB}$, ...), $\equiv_X$ denotes equality, $\equiv_{\mathbb{P}}$ denotes logical equivalence, $\equiv_{A\to B}$ denotes an extensional lift of $\equiv_B$, $\equiv_{A\to\mathbb{P}}$ denotes extensional equivalence, and $\equiv_{\mathsf{ran}}$ denotes range equivalence.

Assuming the existence of surjections $A \to (A \to B)$ may or may not be consistent, depending on the particular equivalence relation. We introduce the notion of **surjection w.r.t.** $\equiv_B$ as $\forall b{:}B.\, \exists a{:}A. f\, a \equiv_B b$. We call a function $f : A \to B$ an **injection w.r.t.** $\equiv_A$ **and** $\equiv_B$ if $\forall a_1 a_2.\, f\, a_1 \equiv_B f\, a_2 \to a_1 \equiv_A a_2$ and a **bijection** if it is an injection and surjection. If we do not specify the equivalence relation we mean equality, i.e. f is an **injection** if $\forall a_1 a_2.\, f\, a_1 = f\, a + 2 \to a_1 = a_2$.

Def. injection

A.4 Predicates

An **isomorphism w.r.t.** $\equiv_A$ **and** $\equiv_B$ between two types A and B is a pair of two functions $f : A \to B$ and $g : B \to A$ such that $\forall a.\, g(f\, a) = a$ and $\forall b.\, f(g\, b) = b$.

Def. isomorphism w.r.t. $\equiv_A$ and $\equiv_B$

Functions (I, R) where $I : X \to Y$ and $R : Y \to \mathbb{O}X$ form a **retraction from** X **to** Y if $\forall x.\, R(I\, x) = \mathsf{Some}\, x$.

Def. retraction from X to Y

Note that if (I, R) is a retraction from X to Y, I is injective.

Let $p : X \to \mathbb{P}$ and $q : Y \to \mathbb{P}$.

The complement $\overline{p} : X \to \mathbb{P}$ of p is defined as $\overline{p} := \lambda x. \neg p x$.

Def. $\overline{p}$

Furthermore we define the sum and the product of predicates analogously to the traditional notions on sets:

Def. $p + q$

$p + q : X + Y \to \mathbb{P} := \lambda s.$ **match** s **with** $\mathsf{inl}\, x \Rightarrow p x \mid \mathsf{inr}\, y \Rightarrow q y$ **end**

Def. $p \times q$

$p \times q : X \times Y \to \mathbb{P} := \lambda(x, y). p x \wedge q y$.

We call a predicate p a **singleton** if $\exists x. p x \wedge \forall y. p y \to x = y$ and a **sub-singleton** if $\forall x y. p x \to p y \to x = y$.

Def. singleton predicate

Def. sub-singleton predicate

A predicate is **inhabited** if $\exists x. p x$ and **non-empty** if $\neg \forall x. \neg p x$.

Note that a predicate is non-empty if and only if $\neg\neg \exists x. p x$, and inhabited predicates are non-empty.

Def. inhabited predicate

A type X is inhabited if $\lambda x{:}X.\, \top$ is inhabited. We then write $\|X\|$. A type X is non-empty of $\lambda x{:}X.\, \top$ is non-empty.

Def. non-empty predicate

Def. $\|X\|$

A predicate p is called **stable predicate** under double negation if $\forall x. \neg\neg p x \to p x$.

Def. stable predicate

Relations are 2-ary predicates $R : X \to Y \to \mathbb{P}$. We call X the **domain** and Y the **range** of R. A relation R is total if $\forall x. \exists y.\, R x y$ and functional if $\forall x y_1 y_2.\, R x y_1 \to R x y_2 \to y_1 = y_2$. We write $R : A \rightsquigarrow B$ if $R : A \to B \to \mathbb{P}$ and R is functional.

Def. domain

Def. range

APPENDIX B

Glossary of synthetic computability

- **Decidable predicate:** → Sec. 4.1, Page 31

  $\mathcal{D}(p\colon X \to \mathbb{P}) := \exists f\colon X \to \mathbb{B}.\forall x.\ px \leftrightarrow f\,x = \mathsf{true}$

- **Discrete type:** → Sec. 4.1, Page 32

  $X \textit{ is discrete} := \exists f\colon X \to X \to \mathbb{B}.\forall x_1 x_2.\ x_1 = x_2 \leftrightarrow f\,x_1 x_2 = \mathsf{true}$

- **Enumerable predicate:** → Sec. 4.2, Page 33

  $\mathcal{E}(p\colon X \to \mathbb{P}) := \exists f\colon \mathbb{N} \to \mathbb{O}X.\forall x.\ px \leftrightarrow \exists n. f\,n = \mathsf{Some}\,x$

- **Enumerable type:** → Sec. 4.2, Page 33

  $X \textit{ is enumerable} := \exists f\colon \mathbb{N} \to \mathbb{O}X.\forall x.\exists n. f\,n = \mathsf{Some}\,x$

- **Parametrically enumerable predicate:** → Sec. 4.2, Page 34

  $\mathcal{E}_{\mathsf{p}}\,(p\colon I \to X \to \mathbb{P}) := \exists(f\colon I \to \mathbb{N} \to \mathbb{O}X).\forall i x.\ p_i x \leftrightarrow \exists n.\ f_i n = \mathsf{Some}\,x$

  We have $\mathcal{E}_{\mathsf{p}}\ p \leftrightarrow \mathcal{E}(\lambda(i,x).\ p_i x)$.

- **Parametrical Enumerability Axiom** → Sec. 6.4, Page 59

  $\mathsf{EA} := \Sigma\varphi\colon \mathbb{N} \to (\mathbb{N} \to \mathbb{O}\mathbb{N}).\forall(p\colon \mathbb{N} \to \mathbb{N} \to \mathbb{P}).$

  $(\exists(f\colon \mathbb{N} \to \mathbb{N} \to \mathbb{O}\mathbb{N}).\forall i.\ f_i \textit{ enumerates } p_i) \to \exists\gamma\colon \mathbb{N} \to \mathbb{N}.\forall i.\ \varphi_i \textit{ enumerates } p_i$

- **Parametrically Universal Enumerator:** → Sec. 6.4, Page 59
  $\varphi\colon \mathbb{N} \to (\mathbb{N} \to \mathbb{O}\mathbb{N})$ such that

  $\forall p\colon \mathbb{N} \to \mathbb{N} \to \mathbb{P}.\ (\exists(f\colon \mathbb{N} \to \mathbb{N} \to \mathbb{O}\mathbb{N}).\forall i.\ f_i \textit{ enumerates } p_i) \to \exists\gamma\colon \mathbb{N} \to \mathbb{N}.\forall i.\ \varphi_i \textit{ enumerates } p_i$

  as assumed in the axiom EA.

- **Semi-decidable predicate:** → Sec. 4.4, Page 37

  $\mathcal{S}(p\colon X \to \mathbb{P}) := \exists f\colon X \to \mathbb{N} \to \mathbb{B}.\forall x.\ px \leftrightarrow \exists n.\ f\,n = \mathsf{true}$

- **Universal Enumerator:** → Sec. 6.4, Page 59

  $\varphi\colon \mathbb{N} \to (\mathbb{N} \to \mathbb{O}\mathbb{N})\ \textit{s.t.}\ \forall p\colon \mathbb{N} \to \mathbb{P}.\ \mathcal{E}p \to \exists c\colon \mathbb{N}.\ \varphi_c \textit{ enumerates } p$

  A parametrically universal enumerator is also universal.

- **Universal Table:** ➔ Sec. 6.4, Page 59

  $$\mathcal{W}_c x := \exists n.\varphi_c n = \mathsf{Some}\, x$$

  where φ is a **universal enumerator** or **parametrically universal enumerator**. We have:

  $$\forall p\colon \mathbb{N} \to \mathbb{N} \to \mathbb{P}.\ \mathcal{E}_{\mathsf{p}}\ p \to \exists \gamma.\forall i x\colon \mathbb{N}.\ p_i x \leftrightarrow \mathcal{W}_{\gamma i} x$$

  if φ is parametrically universal and

  $$\forall p\colon \mathbb{N} \to \mathbb{P}.\ \mathcal{E}\ p \to \exists c.\forall x\colon \mathbb{N}.\ p x \leftrightarrow \mathcal{W}_c x$$

  if φ is universal.