Synthetic Computability in Constructive Type Theory

Yannick Forster

Inria, Gallinette Team, Nantes

Chocola Meeting Lyon, 02.06.2022

received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 101024493.

Work done over the last 7ish years, mostly at Saarland University.

Parts of the work are joint with Dominik Kirst, Gert Smolka, Felix Jahn, Niklas Mück, Nils Lauermann, and Fabian Kunze.

The Coq Undecidability Library has contributions by Dominique Larchey-Wendling, Andrej Dudenhefner, Edith Heiter, Marc Hermes, Johannes Hostert, Dominik Kirst, Mark Koch, Fabian Kunze, Gert Smolka, Simon Spies, Dominik Wehr, Maximilian Wuttke.

Lead questions

How to popularise synthetic computability theory?

Lead questions

How to popularise synthetic computability theory?

How to do constructive reverse analysis of computability theory proofs?

Lead questions

How to popularise synthetic computability theory?

How to do constructive reverse analysis of computability theory proofs?

How to do machine-checked proofs in computability theory?

Lead questions

How to popularise synthetic computability theory?

How to do constructive reverse analysis of computability theory proofs?

How to do machine-checked proofs in computability theory?

Computability Theory

Recipe to write textbooks on computability

1. Introduce favourite model of computation
   1.1 Prove s_n^m theorem (currying)
   1.2 Argue universal program
   1.3 Optional: Introduce a second model and argue equivalence
2. Introduce intuitive computability and Church Turing thesis
3. Develop computability theory relying on Church Turing thesis
   3.1 Undecidability of the halting problem
   3.2 Rice's theorem
   3.3 Reduction theory (Myhill isomorphism theorem, Post's simple and hypersimple sets)
   3.4 Oracle computation and Turing reducibility
4. Prove undecidability of concrete problems (PCP, CFGs)

Recipe to write textbooks on computability

1. Introduce favourite model of computation
   1.1 Prove s_n^m theorem (currying)
   1.2 Argue universal program
   1.3 Optional: Introduce a second model and argue equivalence
2. Introduce intuitive computability and Church Turing thesis
3. Develop computability theory relying on Church Turing thesis
   3.1 Undecidability of the halting problem relying on Church Turing thesis
   3.2 Rice's theorem relying on Church Turing thesis
   3.3 Reduction theory relying on Church Turing thesis
   3.4 Oracle computation relying on Church Turing thesis
4. Prove undecidability (PCP, CFGs) relying on Church Turing thesis

Computability proofs machine-checked in proof assistants

1. Introduce favourite model of computation
   1.1 Prove s_n^m theorem (currying)
   1.2 Argue universal program
   1.3 Optional: Introduce a second model and argue equivalence
2. Introduce intuitive computability and Church Turing thesis
3. Develop computability theory relying on Church Turing thesis
   3.1 Undecidability of the halting problem
   3.2 Rice's theorem
   3.3 Reduction theory
   3.4 Oracle computation
4. Prove undecidability (PCP, CFGs)

Computability proofs machine-checked in proof assistants

1. Introduce favourite model of computation ✓
   1.1 Prove s^m_n theorem (currying)
   1.2 Argue universal program
   1.3 Optional: Introduce a second model and argue equivalence
2. Introduce intuitive computability and Church Turing thesis
3. Develop computability theory relying on Church Turing thesis
   3.1 Undecidability of the halting problem
   3.2 Rice's theorem
   3.3 Reduction theory
   3.4 Oracle computation
4. Prove undecidability (PCP, CFGs)

Computability proofs machine-checked in proof assistants

1. Introduce favourite model of computation ✓
   1.1 Prove s_n^m theorem (currying) ✓
   1.2 Argue universal program
   1.3 Optional: Introduce a second model and argue equivalence
2. Introduce intuitive computability and Church Turing thesis
3. Develop computability theory relying on Church Turing thesis
   3.1 Undecidability of the halting problem
   3.2 Rice's theorem
   3.3 Reduction theory
   3.4 Oracle computation
4. Prove undecidability (PCP, CFGs)

Computability proofs machine-checked in proof assistants

1. Introduce favourite model of computation ✓
   1.1 Prove s^m_n theorem (currying) ✓
   1.2 Argue universal program ✓
   1.3 Optional: Introduce a second model and argue equivalence ✓
2. Introduce intuitive computability and Church Turing thesis
3. Develop computability theory relying on Church Turing thesis
   3.1 Undecidability of the halting problem
   3.2 Rice's theorem
   3.3 Reduction theory
   3.4 Oracle computation
4. Prove undecidability (PCP, CFGs)

Computability proofs machine-checked in proof assistants

1. Introduce favourite model of computation ✓
   1.1 Prove s^m_n theorem (currying) ✓
   1.2 Argue universal program ✓
   1.3 Optional: Introduce a second model and argue equivalence ✓
2. ~~Introduce intuitive computability and Church Turing thesis~~
3. Develop computability theory ~~relying on Church Turing thesis~~
   3.1 Undecidability of the halting problem
   3.2 Rice's theorem
   3.3 Reduction theory
   3.4 Oracle computation
4. Prove undecidability (PCP, CFGs)

Computability proofs machine-checked in proof assistants

1. Introduce favourite model of computation ✓
   1.1 Prove s^m_n theorem (currying) ✓
   1.2 Argue universal program ✓
   1.3 Optional: Introduce a second model and argue equivalence ✓
2. ~~Introduce intuitive computability and Church Turing thesis~~
3. Develop computability theory ~~relying on Church Turing thesis~~
   3.1 Undecidability of the halting problem ✓
   3.2 Rice's theorem
   3.3 Reduction theory
   3.4 Oracle computation
4. Prove undecidability (PCP, CFGs)

Computability proofs machine-checked in proof assistants

1. Introduce favourite model of computation ✓
   1.1 Prove s^m_n theorem (currying) ✓
   1.2 Argue universal program ✓
   1.3 Optional: Introduce a second model and argue equivalence ✓
2. ~~Introduce intuitive computability and Church Turing thesis~~
3. Develop computability theory ~~relying on Church Turing thesis~~
   3.1 Undecidability of the halting problem ✓
   3.2 Rice's theorem ✓
   3.3 Reduction theory
   3.4 Oracle computation
4. Prove undecidability (PCP, CFGs)

Computability proofs machine-checked in proof assistants

1. Introduce favourite model of computation ✓
   1.1 Prove s^m_n theorem (currying) ✓
   1.2 Argue universal program ✓
   1.3 Optional: Introduce a second model and argue equivalence ✓
2. ~~Introduce intuitive computability and Church Turing thesis~~
3. Develop computability theory ~~relying on Church Turing thesis~~
   3.1 Undecidability of the halting problem ✓
   3.2 Rice's theorem ✓
   3.3 Reduction theory **?**
   3.4 Oracle computation **?**
4. Prove undecidability (PCP, CFGs) **?**

Theorem V *For every* $m,n \geq 1$, *there exists a recursive function* $s_n{}^m$ *of* $m + 1$ *variables such that for all* $x, y_1, \ldots, y_m$,

$$\lambda z_1 \cdots z_n[\varphi_x^{(m+n)}(y_1, \ldots, y_m, z_1, \ldots, z_n)] = \varphi^{(n)}_{s_n{}^m(x,y_1,\ldots,y_m)}.$$

Proof. Take the case $m = n = 1$. (Proof is analogous for the other cases.) Consider the family of all partial functions of one variable which are expressible as $\lambda z[\varphi_x{}^{(2)}(y,z)]$ for various x and y. Using our standard formal characterization for functions of two variables, we can view this as a new formal characterization for a class of partial recursive functions of one variable. By Part III of the Basic Result, there exists a uniform effective procedure for going from sets of instructions in this new characterization to sets of instructions in the old. Hence, by Church's Thesis, there must be a recursive function f of two variables such that

$$\lambda z[\varphi_x{}^{(2)}(y,z)] = \varphi_{f(x,y)}.$$

This f is our desired $s_1{}^1$.☒

The informal argument by appeal to Church's Thesis and Part III of the Basic Result can be replaced by a formal proof. (Indeed, the functions $s_n{}^m$ can be shown to be primitive recursive.) We refer the reader to Davis [1958] and Kleene [1952]. Theorem V is known as the *s-m-n theorem* and is due to Kleene. Theorem V (together with Church's Thesis) is a tool of great range and power.

THEOREM 1.1. *There is a primitive recursive function* $\gamma(r, y)$ *such that, for* $n \geqq 1$,

$$[r]^A_{1+n}(y, \mathfrak{x}^{(n)}) = [\gamma(r, y)]_n{}^A(\mathfrak{x}^{(n)}).$$

Intuitively, this result may be interpreted, for $A = \phi$, $n = 1$, as declaring the existence of an algorithm[1] by means of which, given any Turing machine Z and number m, a Turing machine Z_m can be found such that

$$\Psi_{Z^{(2)}}(m, x) = \Psi_{Z_m}(x).$$

Now it is clear that there exist Turing machines Z_m satisfying this last relation since, for each fixed m, $\Psi_{Z^{(2)}}(m, x)$ is certainly a partial recursive function of x. Hence, the content of our theorem (in this special case) is that Z_m can be found effectively in terms of Z and m. However, such a Z_m can readily be described as a Turing machine which, beginning at $\alpha = q_1 1^{x+1}$, proceeds to print $\bar{m} = 1^{m+1}$ to the left, eventually arriving at $\beta = q_N 1^{m+1} B 1^{x+1}$, and then proceeds to act like Z when confronted with

[1] Actually, an algorithm given by a *primitive* recursive function.

$q_1 1^{m+1} B 1^{x+1}$. As the general case does not differ essentially from this special case, all that is required for a formal proof is a detailed construction of Z_m and a careful consideration of the Gödel numbers. The reader who wishes to omit the tedious details, and simply accept the result, may well do so.

PROOF OF THEOREM 1.1. For each value of y, let W_y be the Turing machine consisting of the following quadruples:

$$\begin{array}{l} q_1\ 1\ L\ q_1 \\ q_1\ B\ L\ q_2 \\ \left.\begin{array}{l} q_{i+1}\ B\ 1\ q_{i+1} \\ q_{i+1}\ 1\ L\ q_{i+2} \end{array}\right\} 1 \leqq i \leqq y \\ q_{y+2}\ B\ 1\ q_{y+3}. \end{array}$$

Then, with respect to W_y,

$$\begin{aligned} q_1(\overline{\mathfrak{x}^{(n)}}) &\rightarrow q_1 B(\overline{\mathfrak{x}^{(n)}}) \\ &\rightarrow q_2 BB(\overline{\mathfrak{x}^{(n)}}) \\ &\rightarrow \cdots \\ &\rightarrow q_{y+3}(\overline{y, \mathfrak{x}^{(n)}}). \end{aligned}$$

Let r be a Gödel number of a Turing machine Z, and let

$$Z_y = W_y \cup Z^{(y+2)}.\dagger$$

Then, since the quadruples of $Z^{(y+2)}$ have precisely the same effect on $q_{y+3}(\overline{y, \mathfrak{x}^{(n)}})$ that those of Z have on $q_1(\overline{y, \mathfrak{x}^{(n)}})$, we have

$$\Psi^{(n)}_{Z_y;A}(\mathfrak{x}^{(n)}) = \Psi^{(1+n)}_Z(y, \mathfrak{x}^{(n)}) = [r]^A_{1+n}(y, \mathfrak{x}^{(n)}). \tag{1}$$

We now proceed to evaluate one of the Gödel numbers of Z_y as a function of r and y. The Gödel numbers of the quadruples that make up W_y are as follows:[1]

$$\begin{aligned} a &= \text{gn}\ (q_1\ 1\ L\ q_1) = 2^9 \cdot 3^{11} \cdot 5^5 \cdot 7^9, \\ b &= \text{gn}\ (q_1\ B\ L\ q_2) = 2^9 \cdot 3^7 \cdot 5^5 \cdot 7^{13}, \\ c(i) &= \text{gn}\ (q_{i+1}\ B\ 1\ q_{i+1}) = 2^{4i+9} \cdot 3^7 \cdot 5^{11} \cdot 7^{4i+9},\ 1 \leqq i \leqq y, \\ d(i) &= \text{gn}\ (q_{i+1}\ 1\ L\ q_{i+2}) = 2^{4i+9} \cdot 3^{11} \cdot 5^5 \cdot 7^{4i+13},\ 1 \leqq i \leqq y, \\ e(y) &= \text{gn}\ (q_{y+2}\ B\ 1\ q_{i+3}) = 2^{4y+13} \cdot 3^7 \cdot 5^{11} \cdot 7^{4y+17}. \end{aligned}$$

Thus, if we let

$$\varphi(y) = 2^a \cdot 3^b \cdot 5^{e(y)} \cdot \prod_{i=1}^{y} [\text{Pr}\ (i+3)^{c(i)}\ \text{Pr}\ (i+y+3)^{d(i)}],$$

then $\varphi(y)$ is a primitive recursive function, and, for each y, $\varphi(y)$ is a Gödel number of W_y.

We recall that the predicate IC (x), which is true if and only if x is the number associated with an internal configuration q_i, is primitive recursive, since

$$\text{IC}\ (x) \leftrightarrow \bigvee_{y=0}^{x} (x = 4y + 9).$$

Hence, the function $\iota(x)$, which is 1 when x is the number associated with a q_i and 0 otherwise, is primitive recursive. If h is the Gödel number of a quadruple, then the Gödel number of the quadruple obtained from this one by replacing each q_i by q_{i+y+2} is

$$f(h, y) = 2^{1\,\text{Gl}\,h+4y+8} \cdot 3^{2\,\text{Gl}\,h} \cdot 5^{3\,\text{Gl}\,h+(4y+8)\iota(3\,\text{Gl}\,h)} \cdot 7^{4\,\text{Gl}\,h+4y+8}.$$

Here, $f(h, y)$ is primitive recursive. Hence, if we let

$$\theta(r, y) = \prod_{i=1}^{\mathfrak{L}(r)} \text{Pr}\ (i)^{f(i\,\text{Gl}\,r, y)},$$

then $\theta(r, y)$ is a primitive recursive function and, for each y, $\theta(r, y)$ is a Gödel number of $Z^{(y+2)}$.

Let $\tau(x) = 1$ if x is a Gödel number of a Turing machine; 0, otherwise. Then, by (11) of Chap. 4, Sec. 1, $\tau(x)$ is primitive recursive. Finally, let

$$\gamma(r, y) = (\varphi(y) * \theta(r, y))\tau(r).$$

Then $\gamma(r, y)$ is a primitive recursive function and, for each y, $\gamma(r, y)$ is a Gödel number of Z_y. Hence, by (1),

$$[\gamma(r, y)]_n{}^A(\mathfrak{x}^{(n)}) = [r]^A_{1+n}(y, \mathfrak{x}^{(n)}). \tag{2}$$

It remains only to consider the case where r is not a Gödel number of a Turing machine. In that case, $\gamma(r, y)$, as defined above, is 0 and, thus, is itself not the Gödel number of a Turing machine; so (2) remains correct.[1]

THEOREM 1.2 (Kleene's Iteration Theorem[2]). *For each* m *there is a primitive recursive function* $S^m(r, \mathfrak{y}^{(m)})$ *such that, for* $n \geqq 1$,

$$[r]^A_{m+n}(\mathfrak{y}^{(m)}, \mathfrak{x}^{(n)}) = [S^m(r, \mathfrak{y}^{(m)})]_n{}^A(\mathfrak{x}^{(n)}).$$

Note that Theorem 1.1 is simply Theorem 1.2 with $m = 1$.

```isabelle
section ‹The $s$-$m$-$n$ theorem›

text ‹For all $m, n > 0$ there is an $(m + 1)$-ary primitive recursive
function $s^m_n$ with
\[
  \varphi_p^{(m + n)}(c_1, \dots,c_m, x_1, \dots, x_n) =
  \varphi_{s^m_n(p, c_1, \dots,c_m)}^{(n)}(x_1, \dots, x_n)
\]
for all $p, c_1, \ldots, c_m, x_1, \ldots, x_n$. Here, $\varphi^{(n)}$ is a
function universal for $n$-ary partial recursive functions, which we will
represent by @{term "r_universal n"}›

text ‹The $s^m_n$ functions compute codes of functions. We start simple:
computing codes of the unary constant functions.›

fun code_const1 :: "nat ⇒ nat" where
  "code_const1 0 = 0"
| "code_const1 (Suc c) = quad_encode 3 1 1 (singleton_encode (code_const1 c))"

lemma code_const1: "code_const1 c = encode (r_const c)"
  by (induction c) simp_all

definition "r_code_const1_aux ≡
  Cn 3 r_prod_encode
    [r_constn 2 3,
      Cn 3 r_prod_encode
        [r_constn 2 1,
          Cn 3 r_prod_encode
            [r_constn 2 1, Cn 3 r_singleton_encode [Id 3 1]]]]"

lemma r_code_const1_aux_prim: "prim_recfn 3 r_code_const1_aux"
  by (simp_all add: r_code_const1_aux_def)

lemma r_code_const1_aux:
  "eval r_code_const1_aux [i, r, c] ↓= quad_encode 3 1 1 (singleton_encode r)"
  by (simp add: r_code_const1_aux_def)

definition "r_code_const1 ≡ r_shrink (Pr 1 Z r_code_const1_aux)"

lemma r_code_const1_prim: "prim_recfn 1 r_code_const1"
  by (simp_all add: r_code_const1_def r_code_const1_aux_prim)

lemma r_code_const1: "eval r_code_const1 [c] ↓= code_const1 c"
proof -
  let ?h = "Pr 1 Z r_code_const1_aux"
  have "eval ?h [c, x] ↓= code_const1 c" for x
    using r_code_const1_aux r_code_const1_def
    by (induction c) (simp_all add: r_code_const1_aux_prim)
  then show ?thesis by (simp add: r_code_const1_def r_code_const1_aux_prim)
qed

text ‹Functions that compute codes of higher-arity constant functions:›

definition code_constn :: "nat ⇒ nat ⇒ nat" where
  "code_constn n c ≡
    if n = 1 then code_const1 c
    else quad_encode 3 n (code_const1 c) (singleton_encode (triple_encode 2 n 0))"

lemma code_constn: "code_constn (Suc n) c = encode (r_constn n c)"
  unfolding code_constn_def using code_const1 r_constn_def
  by (cases "n = 0") simp_all

definition r_code_constn :: "nat ⇒ recf" where
  "r_code_constn n ≡
     if n = 1 then r_code_const1
     else
       Cn 1 r_prod_encode
        [r_const 3,
         Cn 1 r_prod_encode
          [r_const n,
           Cn 1 r_prod_encode
            [r_code_const1,
             Cn 1 r_singleton_encode
              [Cn 1 r_prod_encode
                [r_const 2, Cn 1 r_prod_encode [r_const n, Z]]]]]]"

lemma r_code_constn_prim: "prim_recfn 1 (r_code_constn n)"
  by (simp_all add: r_code_constn_def r_code_const1_prim)

lemma r_code_constn: "eval (r_code_constn n) [c] ↓= code_constn n c"
  by (auto simp add: r_code_constn_def r_code_const1 code_constn_def r_code_const1_prim)

text ‹Computing codes of $m$-ary projections:›

definition code_id :: "nat ⇒ nat ⇒ nat" where
  "code_id m n ≡ triple_encode 2 m n"

lemma code_id: "encode (Id m n) = code_id m n"
  unfolding code_id_def by simp

text ‹The functions $s^m_n$ are represented by the following function.
The value $m$ corresponds to the length of @{term "cs"}.›

definition smn :: "nat ⇒ nat ⇒ nat list ⇒ nat" where
  "smn n p cs ≡ quad_encode
     3
     n
     (encode (r_universal (n + length cs)))
     (list_encode (code_constn n p # map (code_constn n) cs @ map (code_id n) [0..<n]))"

lemma smn:
  assumes "n > 0"
  shows "smn n p cs = encode
   (Cn n
     (r_universal (n + length cs))
     (r_constn (n - 1) p # map (r_constn (n - 1)) cs @ (map (Id n) [0..<n])))"
proof -
  let ?p = "r_constn (n - 1) p"
  let ?gs1 = "map (r_constn (n - 1)) cs"
  let ?gs2 = "map (Id n) [0..<n]"
  let ?gs = "?p # ?gs1 @ ?gs2"
  have "map encode ?gs1 = map (code_constn n) cs"
    by (intro nth_equalityI; auto; metis code_constn assms Suc_pred)
  moreover have "map encode ?gs2 = map (code_id n) [0..<n]"
    by (rule nth_equalityI) (auto simp add: code_id_def)
  moreover have "encode ?p = code_constn n p"
    using assms code_constn[of "n - 1" p] by simp
  ultimately have "map encode ?gs =
      code_constn n p # map (code_constn n) cs @ map (code_id n) [0..<n]"
    by simp
  then show ?thesis
    unfolding smn_def using assms encode.simps(4) by presburger
qed

text ‹The next function is to help us define @{typ recf}s corresponding
to the $s^m_n$ functions. It maps $m + 1$ arguments $p, c_1, \ldots, c_m$ to
an encoded list of length $m + n + 1$. The list comprises the $m + 1$ codes
of the $n$-ary constants $p, c_1, \ldots, c_m$ and the $n$ codes for all
$n$-ary projections.›

definition r_smn_aux :: "nat ⇒ nat ⇒ recf" where
```

```isabelle
    list_encode (map (code_constn n) (p # cs) @ map (code_id n) [0..<n])"
proof -
  let ?xs = "map (λi. Cn (Suc m) (r_code_constn n) [Id (Suc m) i]) [0..<Suc m]"
  let ?ys = "map (λi. r_constn m (code_id n i)) [0..<n]"
  have len_xs: "length ?xs = Suc m" by simp

  have map_xs: "map (λg. eval g (p # cs)) ?xs = map Some (map (code_constn n) (p # cs))"
  proof (intro nth_equalityI)
    show len: "length (map (λg. eval g (p # cs)) ?xs) =
        length (map Some (map (code_constn n) (p # cs)))"
      by (simp add: assms(2))

    have "map (λg. eval g (p # cs)) ?xs ! i = map Some (map (code_constn n) (p # cs)) ! i"
        if "i < Suc m" for i
    proof -
      have "map (λg. eval g (p # cs)) ?xs ! i = (λg. eval g (p # cs)) (?xs ! i)"
        using len_xs that by (metis nth_map)
      also have "... = eval (Cn (Suc m) (r_code_constn n) [Id (Suc m) i]) (p # cs)"
        using that len_xs
        by (metis (no_types, lifting) add.left_neutral length_map nth_map nth_upt)
      also have "... = eval (r_code_constn n) [the (eval (Id (Suc m) i) (p # cs))]"
        using r_code_constn_prim assms(2) that by simp
      also have "... = eval (r_code_constn n) [(p # cs) ! i]"
        using len that by simp
      finally have "map (λg. eval g (p # cs)) ?xs ! i ↓= code_constn n ((p # cs) ! i)"
        using r_code_constn by simp
      then show ?thesis
        using len_xs len that by (metis length_map nth_map)
    qed
    moreover have "length (map (λg. eval g (p # cs)) ?xs) = Suc m" by simp
    ultimately show "⋀i. i < length (map (λg. eval g (p # cs)) ?xs) ⟹
        map (λg. eval g (p # cs)) ?xs ! i =
        map Some (map (code_constn n) (p # cs)) ! i"
      by simp
  qed
  moreover have "map (λg. eval g (p # cs)) ?ys = map Some (map (code_id n) [0..<n])"
    using assms(2) by (intro nth_equalityI; auto)
  ultimately have "map (λg. eval g (p # cs)) (?xs @ ?ys) =
      map Some (map (code_constn n) (p # cs) @ map (code_id n) [0..<n])"
    by (metis map_append)
  moreover have "map (λx. the (eval x (p # cs))) (?xs @ ?ys) =
      map the (map (λx. eval x (p # cs)) (?xs @ ?ys))"
    by simp
  ultimately have *: "map (λg. the (eval g (p # cs))) (?xs @ ?ys) =
      (map (code_constn n) (p # cs) @ map (code_id n) [0..<n])"
    by simp

  have "∀i<length ?xs. eval (?xs ! i) (p # cs) = map (λg. eval g (p # cs)) ?xs ! i"
    by (metis nth_map)
  then have
    "∀i<length ?xs. eval (?xs ! i) (p # cs) = map Some (map (code_constn n) (p # cs)) ! i"
    using map_xs by simp
  then have "∀i<length ?xs. eval (?xs ! i) (p # cs) ↓"
    using assms map_xs by (metis length_map nth_map option.simps(3))
  then have xs_converg: "∀z∈set ?xs. eval z (p # cs) ↓"
    by (metis in_set_conv_nth)

  have "∀i<length ?ys. eval (?ys ! i) (p # cs) = map (λx. eval x (p # cs)) ?ys ! i"
    by simp
  then have
    "∀i<length ?ys. eval (?ys ! i) (p # cs) = map Some (map (code_id n) [0..<n]) ! i"
    using assms(2) by simp
  then have "∀i<length ?ys. eval (?ys ! i) (p # cs) ↓"
    by simp
  then have "∀z∈set (?xs @ ?ys). eval z (p # cs) ↓"
    using xs_converg by auto
  moreover have "recfn (length (p # cs)) (Cn (Suc m) (r_list_encode (m + n)) (?xs @ ?ys))"
    using assms r_code_constn_prim by auto
  ultimately have "eval (r_smn_aux n m) (p # cs) =
      eval (r_list_encode (m + n)) (map (λg. the (eval g (p # cs))) (?xs @ ?ys))"
    unfolding r_smn_aux_def using assms by simp
  then have "eval (r_smn_aux n m) (p # cs) =
      eval (r_list_encode (m + n)) (map (code_constn n) (p # cs) @ map (code_id n) [0..<n])"
    using * by metis
  moreover have "length (?xs @ ?ys) = Suc (m + n)" by simp
  ultimately show ?thesis
    using r_list_encode * assms(1) by (metis (no_types, lifting) length_map)
qed

text ‹For all $m, n > 0$, the @{typ recf} corresponding to $s^m_n$ is
given by the next function.›

definition r_smn :: "nat ⇒ nat ⇒ recf" where
 "r_smn n m ≡
    Cn (Suc m) r_prod_encode
     [r_constn m 3,
      Cn (Suc m) r_prod_encode
       [r_constn m n,
        Cn (Suc m) r_prod_encode
          [r_constn m (encode (r_universal (n + m))), r_smn_aux n m]]]"

lemma r_smn_prim [simp]: "n > 0 ⟹ prim_recfn (Suc m) (r_smn n m)"
  by (simp_all add: r_smn_def r_smn_aux_prim)

lemma r_smn:
  assumes "n > 0" and "length cs = m"
  shows "eval (r_smn n m) (p # cs) ↓= smn n p cs"
  using assms r_smn_def r_smn_aux smn_def r_smn_aux_prim by simp

lemma map_eval_Some_the:
  assumes "map (λg. eval g xs) gs = map Some ys"
  shows "map (λg. the (eval g xs)) gs = ys"
  using assms
  by (metis (no_types, lifting) length_map nth_equalityI nth_map option.sel)

text ‹The essential part of the $s$-$m$-$n$ theorem: For all $m, n > 0$
the function $s^m_n$ satisfies
\[
  \varphi_p^{(m + n)}(c_1, \dots,c_m, x_1, \dots, x_n) =
  \varphi_{s^m_n(p, c_1, \dots,c_m)}^{(n)}(x_1, \dots, x_n)
\] for all $p, c_i, x_j$.›

lemma smn_lemma:
  assumes "n > 0" and len_cs: "length cs = m" and len_xs: "length xs = n"
  shows "eval (r_universal (m + n)) (p # cs @ xs) =
    eval (r_universal n) ((the (eval (r_smn n m) (p # cs))) # xs)"
proof -
  let ?s = "r_smn n m"
  let ?f = "Cn n
    (r_universal (n + length cs))
    (r_constn (n - 1) p # map (r_constn (n - 1)) cs @ (map (Id n) [0..<n]))"
  have "eval ?s (p # cs) ↓= smn n p cs"
    using assms r_smn by simp
  then have eval_s: "eval ?s (p # cs) ↓= encode ?f"
    by (simp add: assms(1) smn)

  have "recfn n ?f"
    using len_cs assms by auto
  then have *: "eval (r_universal n) ((encode ?f) # xs) = eval ?f xs"
    using r_universal[of ?f n, OF _ len_xs] by simp

  let ?gs = "r_constn (n - 1) p # map (r_constn (n - 1)) cs @ map (Id n) [0..<n]"
```

```isabelle
      show "length (map (λg. the (eval g xs)) ?gs) = length (p # cs @ xs)"
        by (simp add: len_xs)
      have len: "length (map (λg. the (eval g xs)) ?gs) = Suc (m + n)"
        by (simp add: len_cs)
      moreover have "map (λg. the (eval g xs)) ?gs ! i = (p # cs @ xs) ! i"
        if "i < Suc (m + n)" for i
      proof -
        from that consider "i = 0" | "i > 0 ∧ i < Suc m" | "Suc m ≤ i ∧ i < Suc (m + n)"
          using not_le_imp_less by auto
        then show ?thesis
        proof (cases)
          case 1
          then show ?thesis using assms(1) len_xs by simp
        next
          case 2
          then have "?gs ! i = (map (r_constn (n - 1)) cs) ! (i - 1)"
            using len_cs
            by (metis One_nat_def Suc_less_eq Suc_pred length_map
              less_numeral_extra(3) nth_Cons' nth_append)
          then have "map (λg. the (eval g xs)) ?gs ! i =
              (λg. the (eval g xs)) ((map (r_constn (n - 1)) cs) ! (i - 1))"
            using len by (metis length_map nth_map that)
          also have "... = the (eval ((r_constn (n - 1) (cs ! (i - 1)))) xs)"
            using 2 len_cs by auto
          also have "... = cs ! (i - 1)"
            using r_constn len_xs assms(1) by simp
          also have "... = (p # cs @ xs) ! i"
            using 2 len_cs
            by (metis diff_Suc_1 less_Suc_eq_0_disj less_numeral_extra(3) nth_Cons' nth_append)
          finally show ?thesis .
        next
          case 3
          then have "?gs ! i = (map (Id n) [0..<n]) ! (i - Suc m)"
            using len_cs
            by (simp; metis (no_types, lifting) One_nat_def Suc_less_eq add_leE
              plus_1_eq_Suc diff_diff_left length_map not_le nth_append
              ordered_cancel_comm_monoid_diff_class.add_diff_inverse)
          then have "map (λg. the (eval g xs)) ?gs ! i =
              (λg. the (eval g xs)) ((map (Id n) [0..<n]) ! (i - Suc m))"
            using len by (metis length_map nth_map that)
          also have "... = the (eval ((Id n (i - Suc m))) xs)"
            using 3 len_cs by auto
          also have "... = xs ! (i - Suc m)"
            using len_xs 3 by auto
          also have "... = (p # cs @ xs) ! i"
            using len_cs len_xs 3
            by (metis diff_Suc_1 diff_diff_left less_Suc_eq_0_disj not_le nth_Cons'
              nth_append plus_1_eq_Suc)
          finally show ?thesis .
        qed
      qed
      ultimately show "map (λg. the (eval g xs)) ?gs ! i = (p # cs @ xs) ! i"
          if "i < length (map (λg. the (eval g xs)) ?gs)" for i
        using that by simp
    qed
    ultimately show ?thesis by simp
  qed

theorem smn_theorem:
  assumes "n > 0"
  shows "∃s. prim_recfn (Suc m) s ∧
    (∀p cs xs. length cs = m ∧ length xs = n ⟶
        eval (r_universal (m + n)) (p # cs @ xs) =
        eval (r_universal n) ((the (eval s (p # cs))) # xs))"
  using smn_lemma exI[of _ "r_smn n m"] assms by simp
```

Is there a need for machine-checked computability proofs?

Is there a need for machine-checked computability proofs?

1932 Gödel claims without proof that his decidability proof for the $[\exists^*\forall^2\exists^*, \text{all}, (0)]$ fragment of FOL could be extended to include equality.

... Lots of results depend on Gödel's claim.

Is there a need for machine-checked computability proofs?

1932 Gödel claims without proof that his decidability proof for the $[\exists^*\forall^2\exists^*, \text{all}, (0)]$ fragment of FOL could be extended to include equality.

... Lots of results depend on Gödel's claim.

1984 Goldfarb proves the undecidability of this fragment.

Is there a need for machine-checked computability proofs?

1932 Gödel claims without proof that his decidability proof for the $[\exists^*\forall^2\exists^*, \text{all}, (0)]$ fragment of FOL could be extended to include equality.

... Lots of results depend on Gödel's claim.

1984 Goldfarb proves the undecidability of this fragment.

1988 Kfoury, Tiuryn, and Urzyczyn prove the decidability of semi-unification.

Is there a need for machine-checked computability proofs?

1932 Gödel claims without proof that his decidability proof for the $[\exists^*\forall^2\exists^*, \text{all}, (0)]$ fragment of FOL could be extended to include equality.

… Lots of results depend on Gödel's claim.

1984 Goldfarb proves the undecidability of this fragment.

1988 Kfoury, Tiuryn, and Urzyczyn prove the decidability of semi-unification.

1993 Kfoury, Tiuryn, and Urzyczyn prove the *un*decidability of semi-unification.

Is there a need for machine-checked computability proofs?

- 1932 Gödel claims without proof that his decidability proof for the $[\exists^*\forall^2\exists^*, \text{all}, (0)]$ fragment of FOL could be extended to include equality.
- … Lots of results depend on Gödel's claim.
- 1984 Goldfarb proves the undecidability of this fragment.
- 1988 Kfoury, Tiuryn, and Urzyczyn prove the decidability of semi-unification.
- 1993 Kfoury, Tiuryn, and Urzyczyn prove the *un*decidability of semi-unification.
- 2015 Bimbó proves decidability of the MELL-fragment of linear logic.
- 2019 Straßburger disputes proof, leaving status of problem unresolved.

Machine-checked textbook proofs

Theorem V *For every $m,n \geq 1$, there exists a recursive function $s_n{}^m$ of $m + 1$ variables such that for all $x, y_1, \ldots, y_m$,*

$$\lambda z_1 \cdots z_n[\varphi_x^{(m+n)}(y_1, \ldots, y_m, z_1, \ldots, z_n)] = \varphi^{(n)}_{s_n{}^m(x,y_1,\ldots,y_m)}.$$

Proof. Take the case $m = n = 1$. (Proof is analogous for the other cases.) Consider the family of all partial functions of one variable which are expressible as $\lambda z[\varphi_x{}^{(2)}(y,z)]$ for various x and y. Using our standard formal characterization for functions of two variables, we can view this as a new formal characterization for a class of partial recursive functions of one variable. By Part III of the Basic Result, there exists a uniform effective procedure for going from sets of instructions in this new characterization to sets of instructions in the old. Hence, by Church's Thesis, there must be a recursive function f of two variables such that

$$\lambda z[\varphi_x{}^{(2)}(y,z)] = \varphi_{f(x,y)}.$$

This f is our desired $s_1{}^1$. ☒

The informal argument by appeal to Church's Thesis and Part III of the Basic Result can be replaced by a formal proof. (Indeed, the functions $s_n{}^m$ can be shown to be primitive recursive.) We refer the reader to Davis [1958] and Kleene [1952]. Theorem V is known as the *s-m-n theorem* and is due to Kleene. Theorem V (together with Church's Thesis) is a tool of great range and power.

THEOREM 1.1. *There is a primitive recursive function* $\gamma(r, y)$ *such that, for* $n \geqq 1$,

$$[r]^A_{1+n}(y, \mathfrak{x}^{(n)}) = [\gamma(r, y)]_n{}^A(\mathfrak{x}^{(n)}).$$

Intuitively, this result may be interpreted, for $A = \phi$, $n = 1$, as declaring the existence of an algorithm[1] by means of which, given any Turing machine Z and number m, a Turing machine Z_m can be found such that

$$\Psi_{Z^{(2)}}(m, x) = \Psi_{Z_m}(x).$$

Now it is clear that there exist Turing machines Z_m satisfying this last relation since, for each fixed m, $\Psi_{Z^{(2)}}(m, x)$ is certainly a partial recursive function of x. Hence, the content of our theorem (in this special case) is that Z_m can be found effectively in terms of Z and m. However, such a Z_m can readily be described as a Turing machine which, beginning at $\alpha = q_1 1^{x+1}$, proceeds to print $\bar{m} = 1^{m+1}$ to the left, eventually arriving at $\beta = q_N 1^{m+1} B 1^{x+1}$, and then proceeds to act like Z when confronted with

[1] Actually, an algorithm given by a *primitive* recursive function.

$q_1 1^{m+1} B 1^{x+1}$. As the general case does not differ essentially from this special case, all that is required for a formal proof is a detailed construction of Z_m and a careful consideration of the Gödel numbers. The reader who wishes to omit the tedious details, and simply accept the result, may well do so.

PROOF OF THEOREM 1.1. For each value of y, let W_y be the Turing machine consisting of the following quadruples:

$$\begin{array}{l} q_1\ 1\ L\ q_1 \\ q_1\ B\ L\ q_2 \\ \left.\begin{array}{l} q_{i+1}\ B\ 1\ q_{i+1} \\ q_{i+1}\ 1\ L\ q_{i+2} \end{array}\right\} 1 \leqq i \leqq y \\ q_{y+2}\ B\ 1\ q_{y+3}. \end{array}$$

Then, with respect to W_y,

$$\begin{aligned} q_1(\overline{\mathfrak{x}^{(n)}}) &\rightarrow q_1 B(\overline{\mathfrak{x}^{(n)}}) \\ &\rightarrow q_2 BB(\overline{\mathfrak{x}^{(n)}}) \\ &\rightarrow \cdots \\ &\rightarrow q_{y+3}(\overline{y, \mathfrak{x}^{(n)}}). \end{aligned}$$

Let r be a Gödel number of a Turing machine Z, and let

$$Z_y = W_y \cup Z^{(y+2)}.\dagger$$

Then, since the quadruples of $Z^{(y+2)}$ have precisely the same effect on $q_{y+3}(\overline{y, \mathfrak{x}^{(n)}})$ that those of Z have on $q_1(\overline{y, \mathfrak{x}^{(n)}})$, we have

$$\Psi^{(n)}_{Z_y;A}(\mathfrak{x}^{(n)}) = \Psi^{(1+n)}_{Z}(y, \mathfrak{x}^{(n)}) = [r]^A_{1+n}(y, \mathfrak{x}^{(n)}). \quad (1)$$

We now proceed to evaluate one of the Gödel numbers of Z_y as a function of r and y. The Gödel numbers of the quadruples that make up W_y are as follows:[1]

$$\begin{aligned} a &= \text{gn}\ (q_1\ 1\ L\ q_1) = 2^9 \cdot 3^{11} \cdot 5^5 \cdot 7^9, \\ b &= \text{gn}\ (q_1\ B\ L\ q_2) = 2^9 \cdot 3^7 \cdot 5^5 \cdot 7^{13}, \\ c(i) &= \text{gn}\ (q_{i+1}\ B\ 1\ q_{i+1}) = 2^{4i+9} \cdot 3^7 \cdot 5^{11} \cdot 7^{4i+9},\ 1 \leqq i \leqq y, \\ d(i) &= \text{gn}\ (q_{i+1}\ 1\ L\ q_{i+2}) = 2^{4i+9} \cdot 3^{11} \cdot 5^5 \cdot 7^{4i+13},\ 1 \leqq i \leqq y, \\ e(y) &= \text{gn}\ (q_{y+2}\ B\ 1\ q_{i+3}) = 2^{4y+13} \cdot 3^7 \cdot 5^{11} \cdot 7^{4y+17}. \end{aligned}$$

Thus, if we let

$$\varphi(y) = 2^a \cdot 3^b \cdot 5^{e(y)} \cdot \prod_{i=1}^{y} [\text{Pr}\ (i+3)^{c(i)}\ \text{Pr}\ (i+y+3)^{d(i)}],$$

then $\varphi(y)$ is a primitive recursive function, and, for each y, $\varphi(y)$ is a Gödel number of W_y.

We recall that the predicate IC (x), which is true if and only if x is the number associated with an internal configuration q_i, is primitive recursive, since

$$\text{IC}\ (x) \leftrightarrow \bigvee_{y=0}^{x} (x = 4y + 9).$$

Hence, the function $\iota(x)$, which is 1 when x is the number associated with a q_i and 0 otherwise, is primitive recursive. If h is the Gödel number of a quadruple, then the Gödel number of the quadruple obtained from this one by replacing each q_i by q_{i+y+2} is

$$f(h, y) = 2^{1\,\text{Gl}\,h+4y+8} \cdot 3^{2\,\text{Gl}\,h} \cdot 5^{3\,\text{Gl}\,h+(4y+8)\iota(3\,\text{Gl}\,h)} \cdot 7^{4\,\text{Gl}\,h+4y+8}.$$

Here, $f(h, y)$ is primitive recursive. Hence, if we let

$$\theta(r, y) = \prod_{i=1}^{\mathfrak{L}(r)} \text{Pr}\ (i)^{f(i\,\text{Gl}\,r, y)},$$

then $\theta(r, y)$ is a primitive recursive function and, for each y, $\theta(r, y)$ is a Gödel number of $Z^{(y+2)}$.

Let $\tau(x) = 1$ if x is a Gödel number of a Turing machine; 0, otherwise. Then, by (11) of Chap. 4, Sec. 1, $\tau(x)$ is primitive recursive. Finally, let

$$\gamma(r, y) = (\varphi(y) * \theta(r, y))\tau(r).$$

Then $\gamma(r, y)$ is a primitive recursive function and, for each y, $\gamma(r, y)$ is a Gödel number of Z_y. Hence, by (1),

$$[\gamma(r, y)]_n{}^A(\mathfrak{x}^{(n)}) = [r]^A_{1+n}(y, \mathfrak{x}^{(n)}). \quad (2)$$

It remains only to consider the case where r is not a Gödel number of a Turing machine. In that case, $\gamma(r, y)$, as defined above, is 0 and, thus, is itself not the Gödel number of a Turing machine; so (2) remains correct.[1]

THEOREM 1.2 (Kleene's Iteration Theorem[2]). *For each* m *there is a primitive recursive function* $S^m(r, \mathfrak{y}^{(m)})$ *such that, for* $n \geqq 1$,

$$[r]^A_{m+n}(\mathfrak{y}^{(m)}, \mathfrak{x}^{(n)}) = [S^m(r, \mathfrak{y}^{(m)})]_n{}^A(\mathfrak{x}^{(n)}).$$

Note that Theorem 1.1 is simply Theorem 1.2 with $m = 1$.

```
section ‹The $s$-$m$-$n$ theorem›

text ‹For all $m, n > 0$ there is an $(m + 1)$-ary primitive recursive
function $s^m_n$ with
\[
  \varphi_p^{(m + n)}(c_1, \dots,c_m, x_1, \dots, x_n) =
  \varphi_{s^m_n(p, c_1, \dots,c_m)}^{(n)}(x_1, \dots, x_n)
\]
for all $p, c_1, \ldots, c_m, x_1, \ldots, x_n$. Here, $\varphi^{(n)}$ is a
function universal for $n$-ary partial recursive functions, which we will
represent by @{term "r_universal n"}›

text ‹The $s^m_n$ functions compute codes of functions. We start simple:
computing codes of the unary constant functions.›

fun code_const1 :: "nat ⇒ nat" where
  "code_const1 0 = 0"
| "code_const1 (Suc c) = quad_encode 3 1 1 (singleton_encode (code_const1 c))"

lemma code_const1: "code_const1 c = encode (r_const c)"
  by (induction c) simp_all

definition "r_code_const1_aux ≡
  Cn 3 r_prod_encode
    [r_constn 2 3,
     Cn 3 r_prod_encode
       [r_constn 2 1,
        Cn 3 r_prod_encode
          [r_constn 2 1, Cn 3 r_singleton_encode [Id 3 1]]]]"

lemma r_code_const1_aux_prim: "prim_recfn 3 r_code_const1_aux"
  by (simp_all add: r_code_const1_aux_def)

lemma r_code_const1_aux:
  "eval r_code_const1_aux [i, r, c] ↓= quad_encode 3 1 1 (singleton_encode r)"
  by (simp add: r_code_const1_aux_def)

definition "r_code_const1 ≡ r_shrink (Pr 1 Z r_code_const1_aux)"

lemma r_code_const1_prim: "prim_recfn 1 r_code_const1"
  by (simp_all add: r_code_const1_def r_code_const1_aux_prim)

lemma r_code_const1: "eval r_code_const1 [c] ↓= code_const1 c"
proof -
  let ?h = "Pr 1 Z r_code_const1_aux"
  have "eval ?h [c, x] ↓= code_const1 c" for x
    using r_code_const1_aux r_code_const1_def
    by (induction c) (simp_all add: r_code_const1_aux_prim)
  then show ?thesis by (simp add: r_code_const1_def r_code_const1_aux_prim)
qed

text ‹Functions that compute codes of higher-arity constant functions:›

definition code_constn :: "nat ⇒ nat ⇒ nat" where
  "code_constn n c ≡
    if n = 1 then code_const1 c
    else quad_encode 3 n (code_const1 c) (singleton_encode (triple_encode 2 n 0))"

lemma code_constn: "code_constn (Suc n) c = encode (r_constn n c)"
  unfolding code_constn_def using code_const1 r_constn_def
  by (cases "n = 0") simp_all

definition r_code_constn :: "nat ⇒ recf" where
  "r_code_constn n ≡
     if n = 1 then r_code_const1
     else
       Cn 1 r_prod_encode
        [r_const 3,
         Cn 1 r_prod_encode
          [r_const n,
           Cn 1 r_prod_encode
            [r_code_const1,
             Cn 1 r_singleton_encode
              [Cn 1 r_prod_encode
                [r_const 2, Cn 1 r_prod_encode [r_const n, Z]]]]]]"

lemma r_code_constn_prim: "prim_recfn 1 (r_code_constn n)"
  by (simp_all add: r_code_constn_def r_code_const1_prim)

lemma r_code_constn: "eval (r_code_constn n) [c] ↓= code_constn n c"
  by (auto simp add: r_code_constn_def r_code_const1 code_constn_def r_code_const1_prim)

text ‹Computing codes of $m$-ary projections:›

definition code_id :: "nat ⇒ nat ⇒ nat" where
  "code_id m n ≡ triple_encode 2 m n"

lemma code_id: "encode (Id m n) = code_id m n"
  unfolding code_id_def by simp

text ‹The functions $s^m_n$ are represented by the following function.
The value $m$ corresponds to the length of @{term "cs"}.›

definition smn :: "nat ⇒ nat ⇒ nat list ⇒ nat" where
  "smn n p cs ≡ quad_encode
     3
     n
     (encode (r_universal (n + length cs)))
     (list_encode (code_constn n p # map (code_constn n) cs @ map (code_id n) [0..<n]))"

lemma smn:
  assumes "n > 0"
  shows "smn n p cs = encode
   (Cn n
     (r_universal (n + length cs))
     (r_constn (n - 1) p # map (r_constn (n - 1)) cs @ (map (Id n) [0..<n])))"
proof -
  let ?p = "r_constn (n - 1) p"
  let ?gs1 = "map (r_constn (n - 1)) cs"
  let ?gs2 = "map (Id n) [0..<n]"
  let ?gs = "?p # ?gs1 @ ?gs2"
  have "map encode ?gs1 = map (code_constn n) cs"
    by (intro nth_equalityI; auto; metis code_constn assms Suc_pred)
  moreover have "map encode ?gs2 = map (code_id n) [0..<n]"
    by (rule nth_equalityI) (auto simp add: code_id_def)
  moreover have "encode ?p = code_constn n p"
    using assms code_constn[of "n - 1" p] by simp
  ultimately have "map encode ?gs =
      code_constn n p # map (code_constn n) cs @ map (code_id n) [0..<n]"
    by simp
  then show ?thesis
    unfolding smn_def using assms encode.simps(4) by presburger
qed

text ‹The next function is to help us define @{typ recf}s corresponding
to the $s^m_n$ functions. It maps $m + 1$ arguments $p, c_1, \ldots, c_m$ to
an encoded list of length $m + n + 1$. The list comprises the $m + 1$ codes
of the $n$-ary constants $p, c_1, \ldots, c_m$ and the $n$ codes for all
$n$-ary projections.›

definition r_smn_aux :: "nat ⇒ nat ⇒ recf" where
```

```
    list_encode (map (code_constn n) (p # cs) @ map (code_id n) [0..<n])"
proof -
  let ?xs = "map (λi. Cn (Suc m) (r_code_constn n) [Id (Suc m) i]) [0..<Suc m]"
  let ?ys = "map (λi. r_constn m (code_id n i)) [0..<n]"
  have len_xs: "length ?xs = Suc m" by simp

  have map_xs: "map (λg. eval g (p # cs)) ?xs = map Some (map (code_constn n) (p # cs))"
  proof (intro nth_equalityI)
    show len: "length (map (λg. eval g (p # cs)) ?xs) =
        length (map Some (map (code_constn n) (p # cs)))"
      by (simp add: assms(2))

    have "map (λg. eval g (p # cs)) ?xs ! i = map Some (map (code_constn n) (p # cs)) ! i"
      if "i < Suc m" for i
    proof -
      have "map (λg. eval g (p # cs)) ?xs ! i = (λg. eval g (p # cs)) (?xs ! i)"
        using len_xs that by (metis nth_map)
      also have "... = eval (Cn (Suc m) (r_code_constn n) [Id (Suc m) i]) (p # cs)"
        using that len_xs
        by (metis (no_types, lifting) add.left_neutral length_map nth_map nth_upt)
      also have "... = eval (r_code_constn n) [the (eval (Id (Suc m) i) (p # cs))]"
        using r_code_constn_prim assms(2) that by simp
      also have "... = eval (r_code_constn n) [(p # cs) ! i]"
        using len that by simp
      finally have "map (λg. eval g (p # cs)) ?xs ! i ↓= code_constn n ((p # cs) ! i)"
        using r_code_constn by simp
      then show ?thesis
        using len_xs len that by (metis length_map nth_map)
    qed
    moreover have "length (map (λg. eval g (p # cs)) ?xs) = Suc m" by simp
    ultimately show "⋀i. i < length (map (λg. eval g (p # cs)) ?xs) ⟹
        map (λg. eval g (p # cs)) ?xs ! i =
        map Some (map (code_constn n) (p # cs)) ! i"
      by simp
  qed
  moreover have "map (λg. eval g (p # cs)) ?ys = map Some (map (code_id n) [0..<n])"
    using assms(2) by (intro nth_equalityI; auto)
  ultimately have "map (λg. eval g (p # cs)) (?xs @ ?ys) =
      map Some (map (code_constn n) (p # cs) @ map (code_id n) [0..<n])"
    by (metis map_append)
  moreover have "map (λx. the (eval x (p # cs))) (?xs @ ?ys) =
      map the (map (λx. eval x (p # cs)) (?xs @ ?ys))"
    by simp
  ultimately have *: "map (λg. the (eval g (p # cs))) (?xs @ ?ys) =
      (map (code_constn n) (p # cs) @ map (code_id n) [0..<n])"
    by simp

  have "∀i<length ?xs. eval (?xs ! i) (p # cs) = map (λg. eval g (p # cs)) ?xs ! i"
    by (metis nth_map)
  then have
    "∀i<length ?xs. eval (?xs ! i) (p # cs) = map Some (map (code_constn n) (p # cs)) ! i"
    using map_xs by simp
  then have "∀i<length ?xs. eval (?xs ! i) (p # cs) ↓"
    using assms map_xs by (metis length_map nth_map option.simps(3))
  then have xs_converg: "∀z∈set ?xs. eval z (p # cs) ↓"
    by (metis in_set_conv_nth)

  have "∀i<length ?ys. eval (?ys ! i) (p # cs) = map (λx. eval x (p # cs)) ?ys ! i"
    by simp
  then have
    "∀i<length ?ys. eval (?ys ! i) (p # cs) = map Some (map (code_id n) [0..<n]) ! i"
    using assms(2) by simp
  then have "∀i<length ?ys. eval (?ys ! i) (p # cs) ↓"
    by simp
  then have "∀z∈set (?xs @ ?ys). eval z (p # cs) ↓"
    using xs_converg by auto
  moreover have "recfn (length (p # cs)) (Cn (Suc m) (r_list_encode (m + n)) (?xs @ ?ys))"
    using assms r_code_constn_prim by auto
  ultimately have "eval (r_smn_aux n m) (p # cs) =
      eval (r_list_encode (m + n)) (map (λg. the (eval g (p # cs))) (?xs @ ?ys))"
    unfolding r_smn_aux_def using assms by simp
  then have "eval (r_smn_aux n m) (p # cs) =
      eval (r_list_encode (m + n)) (map (code_constn n) (p # cs) @ map (code_id n) [0..<n])"
    using * by metis
  moreover have "length (?xs @ ?ys) = Suc (m + n)" by simp
  ultimately show ?thesis
    using r_list_encode * assms(1) by (metis (no_types, lifting) length_map)
qed

text ‹For all $m, n > 0$, the @{typ recf} corresponding to $s^m_n$ is
given by the next function.›

definition r_smn :: "nat ⇒ nat ⇒ recf" where
 "r_smn n m ≡
    Cn (Suc m) r_prod_encode
     [r_constn m 3,
      Cn (Suc m) r_prod_encode
       [r_constn m n,
        Cn (Suc m) r_prod_encode
          [r_constn m (encode (r_universal (n + m))), r_smn_aux n m]]]"

lemma r_smn_prim [simp]: "n > 0 ⟹ prim_recfn (Suc m) (r_smn n m)"
  by (simp_all add: r_smn_def r_smn_aux_prim)

lemma r_smn:
  assumes "n > 0" and "length cs = m"
  shows "eval (r_smn n m) (p # cs) ↓= smn n p cs"
  using assms r_smn_def r_smn_aux smn_def r_smn_aux_prim by simp

lemma map_eval_Some_the:
  assumes "map (λg. eval g xs) gs = map Some ys"
  shows "map (λg. the (eval g xs)) gs = ys"
  using assms
  by (metis (no_types, lifting) length_map nth_equalityI nth_map option.sel)

text ‹The essential part of the $s$-$m$-$n$ theorem: For all $m, n > 0$
the function $s^m_n$ satisfies
\[
  \varphi_p^{(m + n)}(c_1, \dots,c_m, x_1, \dots, x_n) =
  \varphi_{s^m_n(p, c_1, \dots,c_m)}^{(n)}(x_1, \dots, x_n)
\] for all $p, c_i, x_j$.›

lemma smn_lemma:
  assumes "n > 0" and len_cs: "length cs = m" and len_xs: "length xs = n"
  shows "eval (r_universal (m + n)) (p # cs @ xs) =
    eval (r_universal n) ((the (eval (r_smn n m) (p # cs))) # xs)"
proof -
  let ?s = "r_smn n m"
  let ?f = "Cn n
    (r_universal (n + length cs))
    (r_constn (n - 1) p # map (r_constn (n - 1)) cs @ (map (Id n) [0..<n]))"
  have "eval ?s (p # cs) ↓= smn n p cs"
    using assms r_smn by simp
  then have eval_s: "eval ?s (p # cs) ↓= encode ?f"
    by (simp add: assms(1) smn)

  have "recfn n ?f"
    using len_cs assms by auto
  then have *: "eval (r_universal n) ((encode ?f) # xs) = eval ?f xs"
    using r_universal[of ?f n, OF _ len_xs] by simp

  let ?gs = "r_constn (n - 1) p # map (r_constn (n - 1)) cs @ map (Id n) [0..<n]"
```

```
    show "length (map (λg. the (eval g xs)) ?gs) = length (p # cs @ xs)"
      by (simp add: len_xs)
    have len: "length (map (λg. the (eval g xs)) ?gs) = Suc (m + n)"
      by (simp add: len_cs)
    moreover have "map (λg. the (eval g xs)) ?gs ! i = (p # cs @ xs) ! i"
      if "i < Suc (m + n)" for i
    proof -
      from that consider "i = 0" | "i > 0 ∧ i < Suc m" | "Suc m ≤ i ∧ i < Suc (m + n)"
        using not_le_imp_less by auto
      then show ?thesis
      proof (cases)
        case 1
        then show ?thesis using assms(1) len_xs by simp
      next
        case 2
        then have "?gs ! i = (map (r_constn (n - 1)) cs) ! (i - 1)"
          using len_cs
          by (metis One_nat_def Suc_less_eq Suc_pred length_map
            less_numeral_extra(3) nth_Cons' nth_append)
        then have "map (λg. the (eval g xs)) ?gs ! i =
            (λg. the (eval g xs)) ((map (r_constn (n - 1)) cs) ! (i - 1))"
          using len by (metis length_map nth_map that)
        also have "... = the (eval ((r_constn (n - 1) (cs ! (i - 1)))) xs)"
          using 2 len_cs by auto
        also have "... = cs ! (i - 1)"
          using r_constn len_xs assms(1) by simp
        also have "... = (p # cs @ xs) ! i"
          using 2 len_cs
          by (metis diff_Suc_1 less_Suc_eq_0_disj less_numeral_extra(3) nth_Cons' nth_append)
        finally show ?thesis .
      next
        case 3
        then have "?gs ! i = (map (Id n) [0..<n]) ! (i - Suc m)"
          using len_cs
          by (simp; metis (no_types, lifting) One_nat_def Suc_less_eq add_leE
            plus_1_eq_Suc diff_diff_left length_map not_le nth_append
            ordered_cancel_comm_monoid_diff_class.add_diff_inverse)
        then have "map (λg. the (eval g xs)) ?gs ! i =
            (λg. the (eval g xs)) ((map (Id n) [0..<n]) ! (i - Suc m))"
          using len by (metis length_map nth_map that)
        also have "... = the (eval ((Id n (i - Suc m))) xs)"
          using 3 len_cs by auto
        also have "... = xs ! (i - Suc m)"
          using len_xs 3 by auto
        also have "... = (p # cs @ xs) ! i"
          using len_cs len_xs 3
          by (metis diff_Suc_1 diff_diff_left less_Suc_eq_0_disj not_le nth_Cons'
            nth_append plus_1_eq_Suc)
        finally show ?thesis .
      qed
    qed
    ultimately show "map (λg. the (eval g xs)) ?gs ! i = (p # cs @ xs) ! i"
      if "i < length (map (λg. the (eval g xs)) ?gs)" for i
      using that by simp
  qed
  ultimately show ?thesis by simp
qed

theorem smn_theorem:
  assumes "n > 0"
  shows "∃s. prim_recfn (Suc m) s ∧
    (∀p cs xs. length cs = m ∧ length xs = n ⟶
        eval (r_universal (m + n)) (p # cs @ xs) =
        eval (r_universal n) ((the (eval s (p # cs))) # xs))"
  using smn_lemma exI[of _ "r_smn n m"] assms by simp
```

Synthetic mathematics to the rescue

Analytic mathematics

Objects of the logic

model

structures under investigation

Synthetic mathematics to the rescue

Analytic mathematics

Objects of the logic — model — structures under investigation

Synthetic mathematics*

Objects of the logic — are turned into — structures under investigation

via axioms

*only possible in constructive mathematics

Synthetic mathematics to the rescue

Analytic mathematics

Objects of the logic — model — structures under investigation

Synthetic mathematics*

Objects of the logic — are turned into — structures under investigation

via axioms

*only possible in constructive mathematics

Constructive mathematics to the rescue

Church-Turing thesis:

"Every effectively calculable function is μ-recursive."

Kreisel [1965]

Constructive mathematics to the rescue

Church-Turing thesis:

“Every effectively calculable function is μ-recursive.”

as an axiom in constructive mathematics

$$\mathsf{CT} := \forall f : \mathbb{N} \to \mathbb{N}.\ \exists c : \mathbb{N}.\ \forall x.\ \phi_c x \triangleright f x$$

where $\phi_c x$ is the value of the c-th μ-recursive function with input x

Kreisel [1965]

Overview

1. Axiom-free “synthetic” computability
2. The axiom CT and it’s status in Coq
3. Fully Synthetic Computability á la Richman and Bauer
4. Synthetic Computability without choice
5. Results
6. The Coq Library of Undecidability Proofs

Definitions

Decidability

$\exists f : \mathbb{N} \to \mathbb{B}. \forall x.\ px \leftrightarrow fx = \text{true}$
$\land\ f$ *is computable*

$\exists f : \mathbb{N} \to \mathbb{B}. \forall x.\ px \leftrightarrow fx = \text{true}$

Definitions

Decidability

$\exists f : \mathbb{N} \to \mathbb{B}. \forall x.\ px \leftrightarrow fx = \text{true}$
$\quad \wedge\ f \textit{ is computable}$

$\exists f : \mathbb{N} \to \mathbb{B}. \forall x.\ px \leftrightarrow fx = \text{true}$

Semi-decidability

$\exists f : \mathbb{N} \rightharpoonup \mathbb{N}. \forall x.\ px \leftrightarrow fx \downarrow$
$\quad \wedge\ f \textit{ is computable}$

$\exists f : \mathbb{N} \to \mathbb{B}. \forall x.\ px \leftrightarrow fx \downarrow$

Definitions

Decidability

$\exists f : \mathbb{N} \to \mathbb{B}. \forall x.\ px \leftrightarrow fx = \text{true}$
$\land$ *f is computable*

$\exists f : \mathbb{N} \to \mathbb{B}. \forall x.\ px \leftrightarrow fx = \text{true}$

Semi-decidability

$\exists f : \mathbb{N} \rightharpoonup \mathbb{N}. \forall x.\ px \leftrightarrow fx \downarrow$
$\land$ *f is computable*

$\exists f : \mathbb{N} \to \mathbb{B}. \forall x.\ px \leftrightarrow fx \downarrow$

Many-one reducibility

$\exists f : \mathbb{N} \to \mathbb{N}. \forall x.\ px \leftrightarrow q(fx)$
$\land$ *f is computable*

$\exists f : \mathbb{N} \to \mathbb{N}. \forall x.\ px \leftrightarrow q(fx)$

Definitions

Decidability

$\exists f : \mathbb{N} \to \mathbb{B}. \forall x.\ px \leftrightarrow fx = \text{true}$
$\wedge$ *f is computable*

$\exists f : \mathbb{N} \to \mathbb{B}. \forall x.\ px \leftrightarrow fx = \text{true}$

Semi-decidability

$\exists f : \mathbb{N} \rightharpoonup \mathbb{N}. \forall x.\ px \leftrightarrow fx \downarrow$
$\wedge$ *f is computable*

$\exists f : \mathbb{N} \to \mathbb{B}. \forall x.\ px \leftrightarrow fx \downarrow$

Many-one reducibility

$\exists f : \mathbb{N} \to \mathbb{N}. \forall x.\ px \leftrightarrow q(fx)$
$\wedge$ *f is computable*

$\exists f : \mathbb{N} \to \mathbb{N}. \forall x.\ px \leftrightarrow q(fx)$

Enumerability, one-one reducibility, truth-table reducibility, ...

Axiom-free synthetic computability I

Myhill’s isomorphism theorem

jww Felix Jahn and Gert Smolka [TYPES ’22]

CT is inconsistent in classical systems. . .

. . . because the characteristic function of the self-halting problem is not general recursive.

$$fn := \textbf{if } \varphi_n n \downarrow \textbf{ then } 1 \textbf{ else } 0$$

Troelstra and van Dalen [1988]

CT is inconsistent in classical systems. . .

. . . because the characteristic function of the self-halting problem is not general recursive.

$$fn := \textbf{ if } \varphi_n n\downarrow \textbf{ then } 1 \textbf{ else } 0$$

Formally in ZF:

$$f := \{(n,1) \mid \varphi_n n \downarrow\} \cup \{(n,0) \mid \varphi_n n \uparrow\}$$

Now f is a total functional relation because f is . . .

☑ functional

☐ total

Troelstra and van Dalen [1988]

CT is inconsistent in classical systems...

...because the characteristic function of the self-halting problem is not general recursive.

$$fn := \textbf{ if } \varphi_n n \downarrow \textbf{ then } 1 \textbf{ else } 0$$

Formally in ZF:

$$f := \{(n, 1) \mid \varphi_n n \downarrow\} \cup \{(n, 0) \mid \varphi_n n \uparrow\}$$

Now f is a total functional relation because f is ...

☑ functional

☑ total (proof by contradiction)

Troelstra and van Dalen [1988]

CT is inconsistent in classical systems…

…because the characteristic function of the self-halting problem is not general recursive.

$$fn := \textbf{ if } \varphi_n n \downarrow \textbf{ then } 1 \textbf{ else } 0$$

Formally in ZF:

$$f := \{(n, 1) \mid \varphi_n n \downarrow\} \cup \{(n, 0) \mid \varphi_n n \uparrow\}$$

Now f is a total set-theoretic function because f is …

☑ functional

☑ total (proof by contradiction)

Troelstra and van Dalen [1988]

CT is consistent in constructive systems

$$\text{CT} := \forall f : \mathbb{N} \to \mathbb{N}. f \textit{ is general recursive}$$

- Heyting arithmetic, Kleene [1945]
- Bishop’s constructive mathematics / Martin-Löf Type Theory
- HoTT (MLTT + propositional truncation + univalence), Swan and Uemura [2019]
- MLTT, Yamada [2020]

Slogans of (Coq's) Type Theory

Types and functions are native

- Inductive types $\mathbb{N}$, $\mathbb{B}$, $A \times B$ and so on
- The function type $A \to B$ consists exactly of programs in a *total*, strongly typed programming language

Propositions behave constructively

- Propositions are types
- Proofs are programs
- (Total, functional) relations are functions $A \to B \to \mathbb{P}$
- Classical principles are independent:

$$\text{DNE} := \forall P : \mathbb{P}.\ \neg\neg P \to P \qquad \text{LEM} := \forall P : \mathbb{P}.\ P \vee \neg P$$

Slogans of (Coq's) Type Theory CIC

Types and functions are native

- Inductive types $\mathbb{N}$, $\mathbb{B}$, $A \times B$ and so on
- The function type $A \to B$ consists exactly of programs in a *total*, strongly typed programming language

Propositions behave constructively

- Propositions are types in a separate, impredicative universe $\mathbb{P}$
- Proofs are programs, no large eliminations from $\mathbb{P}$ to $\mathbb{T}$
- (Total, functional) relations are functions $A \to B \to \mathbb{P}$
- Classical principles are independent:

$$\mathsf{DNE} := \forall P : \mathbb{P}.\ \neg\neg P \to P \qquad \mathsf{LEM} := \forall P : \mathbb{P}.\ P \lor \neg P$$

CT seems to be admissible in CIC

Meta-theoretically:

For every closed term

$$\vdash_{\mathsf{CIC}} f : \mathbb{N} \to \mathbb{N}$$

one can construct a code c with $\vdash_{\mathsf{CIC}} c : \mathbb{N}$ s.t.

$$\vdash_{\mathsf{CIC}} c \textit{ computes } f$$

Follows from semantic extraction theorem for Coq [Letouzey, 2004]
Mechanised proof using weak call-by-value λ-calculus?

CT is not inconsistent in CIC

$$f n := \textbf{if } \varphi_n n \downarrow \textbf{ then } \text{true } \textbf{else} \text{ false}$$

CT is not inconsistent in CIC

$$fn := \textbf{if } \varphi_n n \downarrow \textbf{ then } \text{true} \textbf{ else } \text{false}$$

decision can not be implemented

CT is not inconsistent in CIC

$$fn := \textbf{ if } \varphi_n n \downarrow \textbf{ then } \text{true} \textbf{ else } \text{false}$$

However, we can define the graph relation $G : \mathbb{N} \to \mathbb{B} \to \mathbb{P}$

$$Gnb := \varphi_n n \downarrow \leftrightarrow b = \text{true}$$

CT is not inconsistent in CIC

$$fn := \textbf{if } \varphi_n n \downarrow \textbf{ then } \text{true} \textbf{ else } \text{false}$$

However, we can define the graph relation $G : \mathbb{N} \to \mathbb{B} \to \mathbb{P}$

$$Gnb := \varphi_n n \downarrow \leftrightarrow b = \text{true}$$

☑ G is functional

☐ G is total

CT is not inconsistent in CIC

$$fn := \textbf{if } \varphi_n n \downarrow \textbf{ then } \text{true } \textbf{else } \text{false}$$

However, we can define the graph relation $G : \mathbb{N} \to \mathbb{B} \to \mathbb{P}$

$$Gnb := \varphi_n n \downarrow \leftrightarrow b = \text{true}$$

☑ G is functional

☑ G is total (using proof by contradiction, i.e. LEM)

Relations to functions: Choice principles

The axiom of choice: “every total relation contains a function”

$$\mathrm{AC}_{A,B} := \forall R : A \to B \to \mathbb{P}.(\forall a.\exists b.\ Rab) \to \exists f : A \to B.\forall a.\ Ra(fa)$$

Relations to functions: Choice principles

The axiom of choice: “every total relation contains a function”

$$\mathrm{AC}_{A,B} := \forall R : A \to B \to \mathbb{P}.(\forall a.\exists b.\ Rab) \to \exists f : A \to B.\forall a.\ Ra(fa)$$

Curry Howard isomorphism:

A proof of $\exists b.pb$ is a pair.

A proof of $\forall a.pa$ is a function.

Relations to functions: Choice principles

The axiom of choice: "every total relation contains a function"

$$\mathrm{AC}_{A,B} := \forall R : A \to B \to \mathbb{P}.(\forall a.\exists b.\ Rab) \to \exists f : A \to B.\forall a.\ Ra(fa)$$

Curry Howard isomorphism:

A proof of $\exists b.pb$ is a pair.

A proof of $\forall a.pa$ is a function.

A proof of $\forall a.\exists b.\ Rab$ is a function returning a pair.

Relations to functions: Choice principles

The axiom of choice: “every total relation contains a function”

$$\mathrm{AC}_{A,B} := \forall R : A \to B \to \mathbb{P}.(\forall a.\exists b.\ Rab) \to \exists f : A \to B.\forall a.\ Ra(fa)$$

Curry Howard isomorphism:

A proof of $\exists b.pb$ is a pair.

A proof of $\forall a.pa$ is a function.

A proof of $\forall a.\exists b.\ Rab$ is a function returning a pair.

- ☑ $\forall p : (\exists a.\ Ba) \to \mathbb{P}.\ (\forall (a : A)(b : Ba).\ p(a, b)) \to \forall (s : \exists a.\ Ba).\ ps$
- ☐ $\forall p : (\exists a.\ Ba) \to \mathbb{T}.\ (\forall (a : A)(b : Ba).\ p(a, b)) \to \forall (s : \exists a.\ Ba).\ ps$
- ☐ $\pi_1 : (\exists a.\ Ba) \to A$

Relations to functions: Choice principles

The axiom of choice: “every total relation contains a function”

$$\mathrm{AC}_{A,B} := \forall R : A \to B \to \mathbb{P}.(\forall a.\exists b.\ Rab) \to \exists f : A \to B.\forall a.\ Ra(fa)$$

Curry Howard isomorphism:

A proof of $\exists b.pb$ is a pair.

A proof of $\forall a.pa$ is a function.

A proof of $\forall a.\exists b.\ Rab$ is a function returning a pair.

☑ $\forall p : (\exists a.\ Ba) \to \mathbb{P}.\ (\forall (a : A)(b : Ba).\ p(a, b)) \to \forall (s : \exists a.\ Ba).\ ps$

☒ $\forall p : (\exists a.\ Ba) \to \mathbb{T}.\ (\forall (a : A)(b : Ba).\ p(a, b)) \to \forall (s : \exists a.\ Ba).\ ps$

☐ $\pi_1 : (\exists a.\ Ba) \to A$

Relations to functions: Choice principles

The axiom of choice: “every total relation contains a function”

$$\mathrm{AC}_{A,B} := \forall R : A \to B \to \mathbb{P}.(\forall a.\exists b.\ Rab) \to \exists f : A \to B.\forall a.\ Ra(fa)$$

Curry Howard isomorphism:

A proof of $\exists b.pb$ is a pair.

A proof of $\forall a.pa$ is a function.

A proof of $\forall a.\exists b.\ Rab$ is a function returning a pair.

☑ $\forall p : (\exists a.\ Ba) \to \mathbb{P}.\ (\forall (a : A)(b : Ba).\ p(a, b)) \to \forall (s : \exists a.\ Ba).\ ps$

✗ $\forall p : (\exists a.\ Ba) \to \mathbb{T}.\ (\forall (a : A)(b : Ba).\ p(a, b)) \to \forall (s : \exists a.\ Ba).\ ps$

✗ $\pi_1 : (\exists a.\ Ba) \to A$

Relations to functions: Choice principles

The axiom of choice: “every total relation contains a function”

$$\mathrm{AC}_{A,B} := \forall R : A \to B \to \mathbb{P}.(\forall a.\exists b.\ Rab) \to \exists f : A \to B.\forall a.\ Ra(fa)$$

Theorem

The law of excluded middle and the axiom of countable choice together are inconsistent with CT*:*

$$\mathrm{LEM} \land \mathrm{AC}_{\mathbb{N},\mathbb{B}} \to \neg\mathrm{CT}$$

Axiom-free synthetic computability II

The following are equivalent: $C\text{-AC}_{A,B} := \forall R : A \to B \to \mathbb{P}. CR \to (\forall a. \exists b. Rab) \to \exists f. \forall a.\ Ra(fa)$

Theorem

$\Sigma^0_1\text{-AC}_{\mathbb{N},\mathbb{B}}$

jww Dominik Kirst and Gert Smolka [CPP '19]

Axiom-free synthetic computability II

The following are equivalent: $C\text{-AC}_{A,B} := \forall R : A \to B \to \mathbb{P}. CR \to (\forall a. \exists b. Rab) \to \exists f. \forall a.\ Ra(fa)$

Theorem

$\Sigma^0_1\text{-AC}_{\mathbb{N},\mathbb{B}}$

Theorem

- $\text{MP} := \forall f : \mathbb{N} \to \mathbb{B}.\ \neg\neg(\exists n.\ fn = \text{true}) \to (\exists n.\ fn = \text{true})$
- $\forall X. \forall p : X \to \mathbb{P}.\ \mathcal{S}p \to \forall x.\ \neg\neg px \to px$
- $\forall X. \forall p : X \to \mathbb{P}.\ \mathcal{S}p \to \mathcal{S}\overline{p} \to \forall x.\ px \vee \neg px$
- $\forall X. \forall p : X \to \mathbb{P}.\ \mathcal{S}p \to \mathcal{S}\overline{p} \to \mathcal{D}p$

jww Dominik Kirst and Gert Smolka [CPP '19]

Which axioms keep CIC computational?

$$\mathsf{LEM} \land \mathsf{AC}_{\mathbb{N},\mathbb{B}} \to \neg\mathsf{CT}$$

- Can one of the assumptions be dropped? (No)
- Can one of the assumptions be weakened? (Yes)
- How much?

Weak(est) classical logical and choice principles

Theorem

$$\begin{array}{c} \mathsf{LEM} \\ \wedge \\ \mathsf{AC}_{\mathbb{N},\mathbb{B}} \end{array} \rightarrow \neg\mathsf{CT}$$

Weak(est) classical logical and choice principles

Theorem

$$\begin{array}{c}\text{LEM} \\ \land \\ \forall R : \mathbb{N} \to \mathbb{B} \to \mathbb{P}.\ (\forall n. \exists\, b.\ Rnb) \to \exists f. \forall n.\ Rn(fn)\end{array} \to \neg \text{CT}$$

Weak(est) classical logical and choice principles

Theorem

$$\begin{aligned}&\text{LEM}\\ &\wedge \qquad\qquad\qquad \to \neg\text{CT}\\ &\forall R : \mathbb{N} \to \mathbb{B} \to \mathbb{P}.\ (\forall n. \exists! b.\ Rnb) \to \exists f. \forall n.\ Rn(fn)\end{aligned}$$

Weak(est) classical logical and choice principles

Theorem

$$\begin{matrix} \text{LEM} \\ \wedge \\ \text{AUC}_{\mathbb{N},\mathbb{B}} \end{matrix} \rightarrow \neg\text{CT}$$

AUC: Axiom of unique choice

Weak(est) classical logical and choice principles

Theorem

$$\begin{array}{c} \forall P : \mathbb{P}.\ P \vee \neg P \\ \wedge \\ \mathsf{AUC}_{\mathbb{N},\mathbb{B}} \end{array} \rightarrow \neg \mathsf{CT}$$

AUC: Axiom of unique choice

Weak(est) classical logical and choice principles

Theorem

$$\forall f : \mathbb{N} \to \mathbb{B}. \quad (\exists n.\, fn = \mathsf{true}) \lor \neg(\exists n.\, fn = \mathsf{true}) \land \mathsf{AUC}_{\mathbb{N},\mathbb{B}} \to \neg\mathsf{CT}$$

AUC: Axiom of unique choice

Weak(est) classical logical and choice principles

Theorem

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

$$\land \qquad \to \neg\mathsf{CT}$$

$$\mathsf{AUC}_{\mathbb{N},\mathbb{B}}$$

AUC: Axiom of unique choice

Weak(est) classical logical and choice principles

Theorem

$$\begin{array}{c} \text{WLPO} \\ \wedge \\ \text{AUC}_{\mathbb{N},\mathbb{B}} \end{array} \to \neg\text{CT}$$

AUC: Axiom of unique choice
WLPO: Weak limited principle of omniscience

Weak(est) classical logical and choice principles

Lemma

WKL $\rightarrow \neg$CT, WKL *is Weak Kőnig's Lemma, proof via Kleene trees*

Weak(est) classical logical and choice principles

Lemma

WKL $\to \neg$CT, WKL *is Weak Kőnig's Lemma, proof via Kleene trees*

$$C\text{-AC}_{A,B} := \forall R : A \to B \to \mathbb{P}.CR \to (\forall a.\exists b.Rab) \to \exists f.\forall a.\, Ra(fa)$$

Theorem

$\Sigma^0_1\text{-AC}_{\mathbb{N},\mathbb{B}}$

Weak(est) classical logical and choice principles

Lemma

WKL $\to \neg$CT, WKL *is Weak Kőnig's Lemma, proof via Kleene trees*

$C\text{-AC}_{A,B} := \forall R : A \to B \to \mathbb{P}. CR \to (\forall a. \exists b. Rab) \to \exists f. \forall a.\ Ra(fa)$

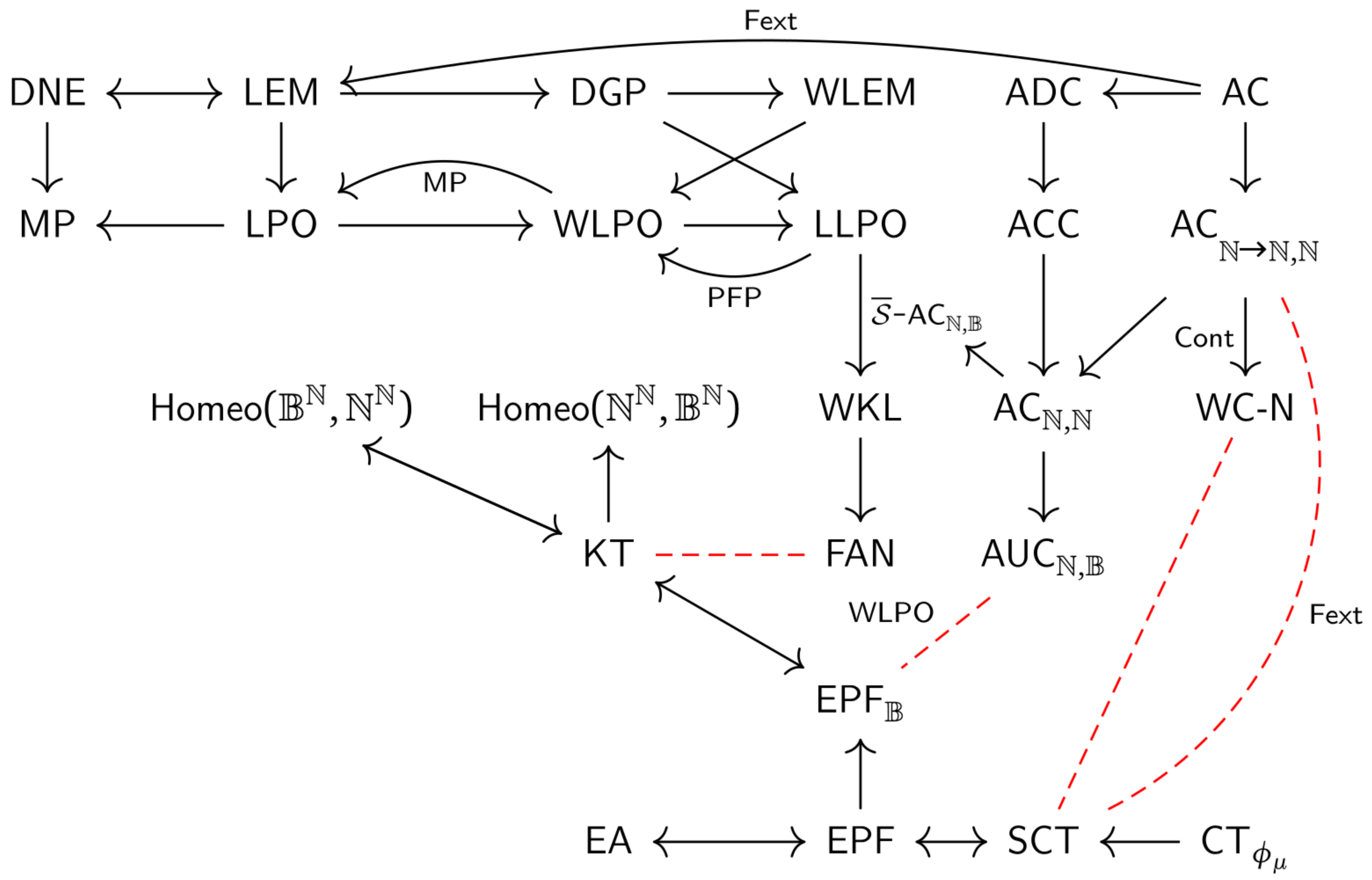
Theorem

$\Sigma^0_1\text{-AC}_{\mathbb{N},\mathbb{B}}$

Theorem

The following are equivalent:

1. WKL
2. LLPO $\wedge$ $\Pi^0_1\text{-AC}_{\mathbb{N},\mathbb{B}}$
3. $\forall R : \mathbb{N} \to \mathbb{B} \to \mathbb{P}.\ R$ is $\Pi^0_1 \to (\forall n. \neg\neg\exists b.\ Rnb) \to \exists f. \forall n.\ Rn(fn)$



Fext
DNE
LEM
DGP
WLEM
ADC
AC
MP
MP
LPO
WLPO
LLPO
ACC
$AC_{\mathbb{N}\to\mathbb{N},\mathbb{N}}$
PFP
$\overline{\mathcal{S}}$-$AC_{\mathbb{N},\mathbb{B}}$
Cont
$\mathrm{Homeo}(\mathbb{B}^{\mathbb{N}}, \mathbb{N}^{\mathbb{N}})$
$\mathrm{Homeo}(\mathbb{N}^{\mathbb{N}}, \mathbb{B}^{\mathbb{N}})$
WKL
$AC_{\mathbb{N},\mathbb{N}}$
WC-N
KT
FAN
$AUC_{\mathbb{N},\mathbb{B}}$
WLPO
Fext
$EPF_{\mathbb{B}}$
EA
EPF
SCT
CT_{ϕ_μ}

Synthetic computability á la Richman

$\phi_c x$ is the value of the c-th μ-recursive function with input x

$$\text{CT} := \quad \forall f : \mathbb{N} \to \mathbb{N}.\ \exists c : \mathbb{N}.\ \forall x.\ \phi_c x \triangleright f x$$

Synthetic computability á la Richman

$$\mathsf{CT}' := \exists \phi. \forall f : \mathbb{N} \to \mathbb{N}.\ \exists c : \mathbb{N}.\ \forall x.\ \phi_c x \triangleright f x$$

Synthetic computability á la Richman, Bridges, and Bauer

$$\mathsf{CT}' := \exists\phi.\forall f : \mathbb{N} \to \mathbb{N}.\ \exists c : \mathbb{N}.\ \forall x.\ \phi_c x \triangleright f x$$

1983 Basic results in computable analysis by Richman

1987 More results in computable analysis by Bridges and Richman

2010 First steps in computability theory by Bauer

Synthetic computability á la Richman, Bridges, and Bauer

$$\mathsf{CT}' := \exists \phi. \forall f : \mathbb{N} \to \mathbb{N}.\ \exists c : \mathbb{N}.\ \forall x.\ \phi_c x \triangleright f x$$

1983 Basic results in computable analysis by Richman

1987 More results in computable analysis by Bridges and Richman

2010 First steps in computability theory by Bauer

All assume the axiom of countable choice, resulting in

Theorem

There is an s^m_n operator for currying.

Synthetic computability á la Richman, Bridges, and Bauer

$$\mathrm{CT}' := \exists \phi. \forall f : \mathbb{N} \to \mathbb{N}.\ \exists c : \mathbb{N}.\ \forall x.\ \phi_c x \triangleright f x$$

1983 Basic results in computable analysis by Richman

1987 More results in computable analysis by Bridges and Richman

2010 First steps in computability theory by Bauer

All assume the axiom of countable choice, resulting in

Theorem

The law of excluded middle is false: $\neg(\forall P : \mathbb{P}.\ P \vee \neg P)$

Synthetic computability á la Richman, Bridges, and Bauer

$$\mathsf{CT}' := \exists \phi. \forall f : \mathbb{N} \to \mathbb{N}.\ \exists c : \mathbb{N}.\ \forall x.\ \phi_c x \triangleright f x$$

1983 Basic results in computable analysis by Richman

1987 More results in computable analysis by Bridges and Richman

2010 First steps in computability theory by Bauer

Bridges and Richman [1987] remark

countable choice can be avoided by postulating an s^m_n operator

Synthetic computability without choice

Assume

1. a (partial) function ϕ
2. universal for $\mathbb{N} \to \mathbb{N}$: $\forall f : \mathbb{N} \to \mathbb{N}. \exists c : \mathbb{N}. \forall x.\ \phi_c x \triangleright fx$,
3. a function $s : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$
4. with the property that $\phi_{s(c,x)} y \equiv \phi_c \langle x, y \rangle$.

Equivalently, using *parametrical* universality

$$\mathsf{SCT} := \exists \phi.\ \forall f : \mathbb{N} \to \mathbb{N} \to \mathbb{N}. \exists \gamma : \mathbb{N} \to \mathbb{N}. \forall i.\ \phi_{\gamma i} \equiv f_i$$

Synthetic computability without choice

Assume

1. a (partial) function ϕ
2. universal for $\mathbb{N} \to \mathbb{N}$: $\forall f : \mathbb{N} \to \mathbb{N}. \exists c : \mathbb{N}. \forall x.\ \phi_c x \triangleright f x$,
3. a function $s : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$
4. with the property that $\phi_{s(c,x)} y \equiv \phi_c \langle x, y \rangle$.

Equivalently, using *parametrical* universality

$$\mathsf{SCT} := \exists \phi.\ \forall f : \mathbb{N} \to \mathbb{N} \to \mathbb{N}. \exists \gamma : \mathbb{N} \to \mathbb{N}. \forall i.\ \phi_{\gamma i} \equiv f_i$$

or using parameterised partial functions $\mathbb{N} \to \mathbb{N} \rightharpoonup \mathbb{N}$ (EPF),

Synthetic computability without choice

Assume

1. a (partial) function ϕ
2. universal for $\mathbb{N} \to \mathbb{N}$: $\forall f : \mathbb{N} \to \mathbb{N}. \exists c : \mathbb{N}. \forall x.\ \phi_c x \triangleright fx$,
3. a function $s : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$
4. with the property that $\phi_{s(c,x)} y \equiv \phi_c \langle x, y \rangle$.

Equivalently, using *parametrical* universality

$$\text{SCT} := \exists \phi. \forall f : \mathbb{N} \to \mathbb{N} \to \mathbb{N}. \exists \gamma : \mathbb{N} \to \mathbb{N}. \forall i.\ \phi_{\gamma i} \equiv f_i$$

or using parameterised partial functions $\mathbb{N} \to \mathbb{N} \rightharpoonup \mathbb{N}$ (EPF),
or using parameterised boolean functions $\mathbb{N} \to \mathbb{N} \rightharpoonup \mathbb{B}$ ($\text{SCT}_{\mathbb{B}}$),

Synthetic computability without choice

Assume

1. a (partial) function ϕ
2. universal for $\mathbb{N} \to \mathbb{N}$: $\forall f : \mathbb{N} \to \mathbb{N}. \exists c : \mathbb{N}. \forall x.\ \phi_c x \triangleright f x$,
3. a function $s : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$
4. with the property that $\phi_{s(c,x)} y \equiv \phi_c \langle x, y \rangle$.

Equivalently, using *parametrical* universality

$$\mathrm{SCT} := \exists \phi.\ \forall f : \mathbb{N} \to \mathbb{N} \to \mathbb{N}. \exists \gamma : \mathbb{N} \to \mathbb{N}. \forall i.\ \phi_{\gamma i} \equiv f_i$$

or using parameterised partial functions $\mathbb{N} \to \mathbb{N} \rightharpoonup \mathbb{N}$ (EPF),
or using parameterised boolean functions $\mathbb{N} \to \mathbb{N} \rightharpoonup \mathbb{B}$ ($\mathrm{SCT}_{\mathbb{B}}$),
or using parametrically enumerable predicates $\mathbb{N} \to \mathbb{N} \to \mathbb{P}$ (EA).

Synthetic computability without choice

Assume

1. a (partial) function ϕ
2. universal for $\mathbb{N} \to \mathbb{N}$: $\forall f : \mathbb{N} \to \mathbb{N}. \exists c : \mathbb{N}. \forall x.\ \phi_c x \triangleright f x$,
3. a function $s : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$
4. with the property that $\phi_{s(c,x)} y \equiv \phi_c \langle x, y \rangle$.

due to strict separation of functions and logic in Coq
the law of excluded middle can be consistently assumed

1. Introduce favourite model of computation
   1.1 Prove s^m_n theorem (currying)
   1.2 Argue universal program
   1.3 Optional: Introduce a second model and argue equivalence
2. Define Church Turing thesis as axiom (SCT, EPF, EA)
3. Develop computability theory relying on axiom
   3.1 Undecidability of the halting problem
   3.2 Rice's theorem
   3.3 Reduction theory (Myhill isomorphism theorem, Post's simple and hypersimple sets)
   3.4 Oracle computation and Turing reducibility
   3.5 Kolmogorov complexity
   3.6 Kleene-Post and Post's theorem
4. Prove undecidability of concrete problems (PCP, CFGs)

1. Introduce favourite model of computation
   1.1 Prove s_n^m theorem (currying)
   1.2 Argue universal program
   1.3 Optional: Introduce a second model and argue equivalence
2. Define Church Turing thesis as axiom (SCT, EPF, EA) ✓
3. Develop computability theory relying on axiom ✓
   3.1 Undecidability of the halting problem ✓
   3.2 Rice's theorem ✓
   3.3 Reduction theory (Myhill isomorphism theorem, Post's simple and hypersimple sets) ✓
   3.4 Oracle computation and Turing reducibility ✓
   3.5 Kolmogorov complexity ✓
   3.6 Kleene-Post and Post's theorem ✓
4. Prove undecidability of concrete problems (PCP, CFGs, needs CT) ✓

Principles in CIC

- Law of excluded middle LEM and Markov’s Principle MP are
  - consistent (important to formalise textbook proofs)

Principles in CIC

- Law of excluded middle LEM and Markov's Principle MP are
  - consistent (important to formalise textbook proofs)
  - but not provable (important for analysing minimal requirements)

Principles in CIC

- Law of excluded middle LEM and Markov's Principle MP are
  - consistent (important to formalise textbook proofs)
  - but not provable (important for analysing minimal requirements)
- Axioms of choice, countable choice, and countable Π^0_1-choice are
  - consistent (nice to know)

Principles in CIC

- Law of excluded middle LEM and Markov's Principle MP are
  - consistent (important to formalise textbook proofs)
  - but not provable (important for analysing minimal requirements)
- Axioms of choice, countable choice, and countable Π^0_1-choice are
  - consistent (nice to know)
  - but not provable (otherwise LEM $\wedge$ CT would be inconsistent)

Principles in CIC

- Law of excluded middle LEM and Markov's Principle MP are
  - consistent (important to formalise textbook proofs)
  - but not provable (important for analysing minimal requirements)
- Axioms of choice, countable choice, and countable Π^0_1-choice are
  - consistent (nice to know)
  - but not provable (otherwise LEM $\wedge$ CT would be inconsistent)
- Axiom of countable Σ^0_1-choice is provable

Principles in CIC

- Law of excluded middle LEM and Markov's Principle MP are
  - consistent (important to formalise textbook proofs)
  - but not provable (important for analysing minimal requirements)
- Axioms of choice, countable choice, and countable Π^0_1-choice are
  - consistent (nice to know)
  - but not provable (otherwise LEM $\wedge$ CT would be inconsistent)
- Axiom of countable Σ^0_1-choice is provable

$\Rightarrow$ enables constructive reverse mathematics for computability

- not too strong (no Π^0_1-choice, LEM, MP)
- just strong enough (countable Σ^0_1-choice)
- This is not the case in (all?) other type theories

Other type theories

- Martin-Löf Type Theory (e.g. Agda) with $\exists x.px := \Sigma x.px$:
  Proves AC, so LLPO $\to \neg$CT.
- Martin-Löf Type Theory (e.g. Agda) with $\exists x.px := \neg\neg\Sigma x.px$:
  Does not prove AC, but Π^0_1-AC$_{\mathbb{N},\mathbb{B}} \to \neg$CT
- Homotopy Type Theory with $\exists x.px := ||\Sigma x.px||$:
  Proves AUC, so WLPO $\to \neg$CT.

Constructive Reverse Mathematics in CIC

Fred Richman:

“Countable choice is a blind spot for constructive mathematicians in much the same way as excluded middle is for classical mathem-aticians.”

Richman [2000, 2001]

Constructive Reverse Mathematics in CIC

Fred Richman:

“Countable choice is a blind spot for constructive mathematicians in much the same way as excluded middle is for classical mathematicians.”

Me:

“CIC is a suitable base system for constructive (reverse) mathematics sensitive to applications of countable choice.”

Richman [2000, 2001]

Three Flavours

- No axioms
  - Morally identify computable functions with functions
  - Can prove results not relying on a universal machine
- With CT as axiom
  - Needs a model of computation
  - Allows proving undecidability of concrete problems
  - Allows talking e.g. about the arithmetical hierarchy
- With SCT as axiom
  - No need for model of computation

Conjecture

The following are consistent in CIC:

- CT (implies in particular SCT)
- LEM (implies in particular MP)
- functional extensionality
- Uniformisation: “Every total relation contains a total functional subrelation.”

Results

Rice's theorem

$$\mathsf{EPF} := \exists \phi. \forall f : \mathbb{N} \to \mathbb{N} \nrightarrow \mathbb{N}. \exists \gamma.\ \forall i x.\ \phi_{\gamma i} x \triangleright f_i x$$

$$\begin{aligned}\mathsf{EA} := \exists \varphi. \forall p : \mathbb{N} \to \mathbb{N} \to \mathbb{P}.\\ (\exists f. \forall i.\ f_i \textit{ enumerates } p_i) \to \exists \gamma. \forall i.\ \varphi_{\gamma i} \textit{ enumerates } p_i\end{aligned}$$

Rice's theorem

$$\text{EPF} := \exists \phi. \forall f : \mathbb{N} \to \mathbb{N} \nrightarrow \mathbb{N}. \exists \gamma.\ \forall ix.\ \phi_{\gamma i} x \triangleright f_i x$$

$$\text{EA} := \exists \varphi. \forall p : \mathbb{N} \to \mathbb{N} \to \mathbb{P}.$$
$$(\exists f. \forall i.\ f_i \text{ enumerates } p_i) \to \exists \gamma. \forall i.\ \varphi_{\gamma i} \text{ enumerates } p_i$$

Theorem

Given EPF *every* $p : (\mathbb{N} \rightharpoonup \mathbb{N}) \to \mathbb{P}$ *is undecidable if it*

1. *is extensional:* $\forall f f' : \mathbb{N} \rightharpoonup \mathbb{N}. (\forall x.\ fx \equiv f'x) \to pf \leftrightarrow pf'$
2. *is non-trivial:* $\exists f_1 f_2.\ pf_1 \wedge \neg pf_2$

Rice's theorem

$$\mathrm{EPF} := \exists \phi. \forall f : \mathbb{N} \to \mathbb{N} \nrightarrow \mathbb{N}. \exists \gamma.\ \forall i x.\ \phi_{\gamma i} x \triangleright f_i x$$

$$\mathrm{EA} := \exists \varphi. \forall p : \mathbb{N} \to \mathbb{N} \to \mathbb{P}.$$
$$(\exists f. \forall i.\ f_i \textit{ enumerates } p_i) \to \exists \gamma. \forall i.\ \varphi_{\gamma i} \textit{ enumerates } p_i$$

Theorem

Given EA *every* $p : (\mathbb{N} \to \mathbb{P}) \to \mathbb{P}$ *is undecidable if it*

1. *is extensional:* $\forall q q' : \mathbb{N} \to \mathbb{P}. (\forall x.\ q x \leftrightarrow q' x) \to p q \leftrightarrow p q'$
2. *is non-trivial:* $\exists q_1 q_2$ *both enumerable.* $p q_1 \wedge \neg p f_2$

$$\mathsf{EPF} := \exists \phi . \forall f : \mathbb{N} \to \mathbb{N} \nrightarrow \mathbb{N} . \exists \gamma .\ \forall i x .\ \phi_{\gamma i} x \triangleright f_i x$$

Lemma

Let ϕ be given as in EPF *and $\gamma : \mathbb{N} \to \mathbb{N}$, then there exists c s.t. $\phi_{\gamma c} \equiv \phi_c$.*

Theorem

Let ϕ be given as in EPF *and $p : \mathbb{N} \to \mathbb{P}$. If p treats elements as codes w.r.t. ϕ and is non-trivial, then p is undecidable.*

Proof.

Let f decide p and let pc_1 and $\neg pc_2$. Define $h_x y :=$ **if** fx *then* $\phi_{c_2} y$ **else** $\phi_{c_1} y$ and let γ via EPF be s.t. $\phi_{\gamma x} \equiv h_x$. Let c be a fixed-point for γ.
Case analysis on fc:

- If fc = true we have $\quad pc$ and $\phi_c \equiv \phi_{\gamma c} \equiv h_c \equiv \phi_{c_2}$. Thus $\quad pc_2$, contradiction.
- If fc = false we have $\neg pc$ and $\phi_c \equiv \phi_{\gamma c} \equiv h_c \equiv \phi_{c_1}$. Thus $\neg pc_1$, contradiction.

□

Simple predicates

Definition (analytic)

A predicate $p : \mathbb{N} \to \mathbb{P}$ is called *simple* if

- it is enumerable,
- its complement is infinite,
- its complement has no enumerable infinite subpredicate.

jww Felix Jahn

Simple predicates

Definition (analytic)

A predicate $p : \mathbb{N} \to \mathbb{P}$ is called *simple* if

- it is enumerable,
- its complement is infinite,
- its complement has no enumerable infinite subpredicate.

Definition

A predicate $p : \mathbb{N} \to \mathbb{P}$ is *infinite* if there exists an injection of type $\mathbb{N} \to \mathbb{N}$ returning only elements in p.

jww Felix Jahn

Simple predicates

Definition (analytic)

A predicate $p : \mathbb{N} \to \mathbb{P}$ is called *simple* if

- it is enumerable,
- its complement is infinite,
- its complement has no enumerable infinite subpredicate.

Definition

A predicate $p : \mathbb{N} \to \mathbb{P}$ is *infinite* if there exists an injection of type $\mathbb{N} \to \mathbb{N}$ returning only elements in p.

Theorem

Every infinite predicate has an enumerable infinite subpredicate.

jww Felix Jahn

Simple predicates

Definition (analytic)

A predicate $p : \mathbb{N} \to \mathbb{P}$ is called *simple* if

- it is enumerable,
- its complement is infinite,
- its complement has no enumerable infinite subpredicate.

Definition

A predicate $p : \mathbb{N} \to \mathbb{P}$ is *infinite* if $\forall n. \exists x > n.\ px$.

Theorem (Meta)

Every definable predicate which can be proved infinite can be proved to have an enumerable subpredicate.

jww Felix Jahn

Simple predicates

Definition (analytic)

A predicate $p : \mathbb{N} \to \mathbb{P}$ is called *simple* if

- it is enumerable,
- its complement is infinite,
- its complement has no enumerable infinite subpredicate.

Definition

A predicate $p : \mathbb{N} \to \mathbb{P}$ is *infinite* if there is no list covering p.

jww Felix Jahn

Kolmogorov complexity

We call a partial function $\mathcal{D} : \mathbb{N} \rightharpoonup \mathbb{N}$ a *description mode*. We call y a description of x if $\mathcal{D}y \triangleright x$. $|n|$ is the length of the bit string representing a number n.

$$\forall y' x.\ \mathcal{D}' y' \triangleright x \to \exists y.\ \mathcal{D}y \triangleright x \land |y| < |y'| + d.$$

$$\mathcal{C}xs := s \text{ is } \boldsymbol{\mu}s.\ \exists y.\ s = |y| \land \mathcal{D}y \triangleright x$$

$$\mathcal{N}(x) := \mathcal{C}x < x$$

Lemma

$\forall x. \neg\neg\exists s.\ \mathcal{C}xs$

Theorem

$\mathcal{N}$ *is simple*

jww Nils Lauermann and Fabian Kunze [ITP '22]

Turing reducibility

Analytic: A μ-recursive functional takes as input an oracle and a number and may compute a number. Theorem by Kleene and Davis:

$$F(\alpha)x \triangleright_\mu y \to \exists L{:}\mathbb{LN}.\ (\forall x \in L.\ \exists y.\ \alpha x \triangleright y) \land \forall \beta.\ (\forall x \in L.\ \alpha x = \beta x) \to F(\beta)x \triangleright_\mu y$$

jww Dominik Kirst [TYPES ’22]

Turing reducibility

$$F(\alpha)x \triangleright_\mu y \to \exists L{:}\mathbb{L}\mathbb{N}.\ (\forall x \in L.\ \exists y.\ \alpha x \triangleright y) \land \forall \beta.\ (\forall x \in L.\ \alpha x = \beta x) \to F(\beta)x \triangleright_\mu y$$

Synthetically, a Turing functional $F{:}(Y \rightsquigarrow \mathbb{B}) \to (X \rightsquigarrow \mathbb{B})$ …

1. …is *continuous* if: $\forall R{:}Y \rightsquigarrow \mathbb{B}. \forall x{:}X. \forall b{:}\mathbb{B}.\ FRxb \to \exists L : \mathbb{L}Y.\ (\forall y \in L. \exists b.\ Ryb) \land$
   $\forall R'{:}Y \rightsquigarrow \mathbb{B}.\ (\forall y \in L. \forall b.\ Ryb \to R'yb) \to FR'xb$
2. …factors through a *computational core* $F'{:}(Y \rightharpoonup \mathbb{B}) \to (X \rightharpoonup \mathbb{B})$ if:

$$\forall f{:}Y \rightharpoonup \mathbb{B}. \forall R{:}Y \rightsquigarrow \mathbb{B}.\ f \textit{ computes } R \to F'f \textit{ computes } FR$$

where $f{:}Z_1 \rightharpoonup Z_2$ *computes* a functional relation $R{:}Z_1 \rightsquigarrow Z_2$ if $\forall xy.\ Rxy \leftrightarrow fx \triangleright y$.

A synthetic Turing reduction from p to $q{:}Y \to \mathbb{P}$ maps the characteristic relation of q to the one of p.

jww Dominik Kirst [TYPES '22]

The arithmetical hierarchy

All first-order logic formulas is equivalent to a formula in prenex normal form if and only if LEM holds.

We can define a predicate $p : \mathbb{N} \to \mathbb{P}$ to be

- Σ_0 and Π_0 if it is expressible as quantor-free arithmetical formula.
- Σ_{n+1} if there is a quantor-free arithmetical formula q with $\forall x.\ px \leftrightarrow \exists \vec{y_1} \forall \vec{y_2} \dots \nabla \vec{y_n}.\ q(x, \vec{y_1}, \vec{y_2}, \dots, \vec{y_n})$
- Π_{n+1} if there is a quantor-free arithmetical formula q with $\forall x.\ px \leftrightarrow \forall \vec{y_1} \exists \vec{y_2} \dots \nabla \vec{y_n} \dots .\ q(x, \vec{y_1}, \vec{y_2}, \dots, \vec{y_n})$

jww Niklas Mück and Dominik Kirst [TYPES ’22]

The arithmetical hierarchy

All first-order logic formulas is equivalent to a formula in prenex normal form if and only if LEM holds.

We can define a predicate $p : \mathbb{N} \to \mathbb{P}$ to be

- Σ_0 and Π_0 if it is expressible as quantor-free arithmetical formula.
- Σ_{n+1} if there is a quantor-free arithmetical formula q with $\forall x.\, px \leftrightarrow \exists \vec{y_1} \forall \vec{y_2} \dots \nabla \vec{y_n}.\, q(x, \vec{y_1}, \vec{y_2}, \dots, \vec{y_n})$
- Π_{n+1} if there is a quantor-free arithmetical formula q with $\forall x.\, px \leftrightarrow \forall \vec{y_1} \exists \vec{y_2} \dots \nabla \vec{y_n} \dots .\, q(x, \vec{y_1}, \vec{y_2}, \dots, \vec{y_n})$

Or replace *quantor-free* by *decidable*.

Theorem

Both definitions are equivalent under CT.

jww Niklas Mück and Dominik Kirst [TYPES '22]

Ever seen this principle?

Markov's Principle

$$\text{MP} := \forall f{:}\mathbb{N}\to\mathbb{B}. \qquad \neg\neg(\exists n.\ fn = \text{true}) \leftrightarrow (\exists n.\ fn = \text{true})$$

Anonymised Markov's Principle

$$\text{AMP} := \forall f{:}\mathbb{N}\to\mathbb{B}.\exists g{:}\mathbb{N}\to\mathbb{B}.\ \neg\neg(\exists n.\ fn = \text{true}) \leftrightarrow (\exists n.\ gn = \text{true})$$

Ever seen this principle?

Markov's Principle

$$\text{MP} := \forall f{:}\mathbb{N}\to\mathbb{B}.\quad \neg\neg(\exists n.\ fn = \text{true}) \leftrightarrow (\exists n.\ fn = \text{true})$$

Anonymised Markov's Principle

$$\text{AMP} := \forall f{:}\mathbb{N}\to\mathbb{B}.\exists g{:}\mathbb{N}\to\mathbb{B}.\ \neg\neg(\exists n.\ fn = \text{true}) \leftrightarrow (\exists n.\ gn = \text{true})$$

Principle of Finite Possibility

$$\text{PFP} := \forall f{:}\mathbb{N}\to\mathbb{B}.\exists g{:}\mathbb{N}\to\mathbb{B}.\quad \neg(\exists n.\ fn = \text{true}) \leftrightarrow (\exists n.\ gn = \text{true})$$

Post's theorem

Let r_n enumerate all continuous functions $F' : (Y \rightharpoonup \mathbb{B}) \to (X \rightharpoonup \mathbb{B})$.

Lemma

There is a Turing functional with core F'.

$$A' := \lambda n.\, \exists R.\, (\forall f.\, R\, f = r_n\, f) \land R\, A\, n\, \mathsf{true}$$

A is semi-decidable relative to B if there is a Turing functional F with

$$\forall n.\, A n \leftrightarrow F\, B\, n\, \mathsf{true}.$$

Theorem (Post)

Assuming LEM*:*

- *A unary predicate A is Σ_{n+1} iff it is semi-deciable relative to $\emptyset^{(n)}$.*
- *If A is Σ_n, then $A \preceq_T \emptyset^{(n)}$.*

jww with Niklas Mück and Dominik Kirst [TYPES '22]

Completeness of first-order logic

Let $\mathcal{T} \vDash \varphi$ be Tarski-style validity of a formula φ under theory $\mathcal{T}$ in all models $\mathcal{M}$ satisfying Peirce's law, where n-ary functions are interpreted as functions $D^n \to D$ and predicates as predicates $D^n \to \mathbb{P}$.
α-completeness for $\alpha : (\text{form} \to \mathbb{P}) \to \mathbb{P}$

$$\forall \mathcal{T} : \text{form} \to \mathbb{P}.\ \alpha(\mathcal{T}) \to \forall \varphi : \text{form}.\ \mathcal{T} \vDash \varphi \to (\exists \Gamma : \text{listform}.\ \Gamma \subset \mathcal{T} \wedge \Gamma \vdash \varphi)$$

- Arbitrary completeness is equivalent to LEM
- $\mathcal{D}$-completeness is equivalent to MP
- $\mathcal{E}$-completeness is equivalent to MP
- finite-completeness is equivalent to $\forall f : \mathbb{N} \to \mathbb{B}.\ \text{computable } f \to \dots$

jww Dominik Kirst and Dominik Wehr [LFCS '20, LOGCOM '21]

Completeness of first-order logic

If we interpret predicates as boolean functions $D^n \to \mathbb{B}$ we have that

- Arbitrary completeness is equivalent to LEM and Weak König's Lemma for arbitrary trees
- $\mathcal{D}$-completeness follows from LEM and Weak König's Lemma for arbitrary trees
- $\mathcal{E}$-completeness is equivalent to ???
- finite-completeness is equivalent to ???

jww Dominik Kirst, Hugo Herbelin

Completeness of first-order logic

If we interpret predicates as boolean functions $D^n \to \mathbb{B}$ we have that

- Arbitrary completeness is equivalent to LEM and Weak König's Lemma for arbitrary trees
- $\mathcal{D}$-completeness follows from LEM and Weak König's Lemma for arbitrary trees *Does* MP *suffice? Equivalence?*
- $\mathcal{E}$-completeness is equivalent to ???
- finite-completeness is equivalent to ???

jww Dominik Kirst, Hugo Herbelin

Completeness of first-order logic

If we interpret predicates as boolean functions $D^n \to \mathbb{B}$ we have that

- Arbitrary completeness is equivalent to LEM and Weak König's Lemma for arbitrary trees
- $\mathcal{D}$-completeness follows from LEM and Weak König's Lemma for arbitrary trees *Does* MP *suffice? Equivalence?*
- $\mathcal{E}$-completeness is equivalent to ???
- finite-completeness is equivalent to ??? WKL *for computable trees is false.*

jww Dominik Kirst, Hugo Herbelin

The Coq Library of Undecidability Proofs

Synthetic undecidability

Analytic definition

$$\mathcal{U}p := \neg\exists f.\ \mu\text{-recursive } f \wedge \ldots$$

Lemma (Analytic)

There is no μ-recursive enumerator for the complement of the halting problem.

Theorem (Analytic)

Given a μ-recursive decider for p, there is a μ-recursive enumerator for the complement of the halting problem:

$$\mathcal{D}p \to \mathcal{E}(\overline{\text{Halt}_{\text{TM}}})$$

Synthetic undecidability

Analytic definition

$$\mathcal{U}p := \neg\exists f.\ \mu\text{-recursive } f \wedge \ldots$$

Lemma (Synthetic)

There is no enumerator for the complement of the halting problem, assuming CT*.*

Theorem (Synthetic)

Given a decider for p, there is an enumerator for the complement of the halting problem:

$$\mathcal{D}p \to \mathcal{E}(\overline{\mathsf{Halt}_{\mathsf{TM}}})$$

Synthetic undecidability

Analytic definition

$$\mathcal{U}p := \neg\exists f.\ \mu\text{-recursive } f \wedge \ldots$$

Lemma (Synthetic)

There is no enumerator for the complement of the halting problem, assuming CT.

Theorem (Synthetic)

Given a decider for p, there is an enumerator for the complement of the halting problem:

$$\mathcal{D}p \to \mathcal{E}(\overline{\mathsf{Halt}_{\mathsf{TM}}})$$

Synthetic undecidability

Analytic definition

$$\mathcal{U}p := \neg\exists f.\ \mu\text{-recursive } f \wedge \dots$$

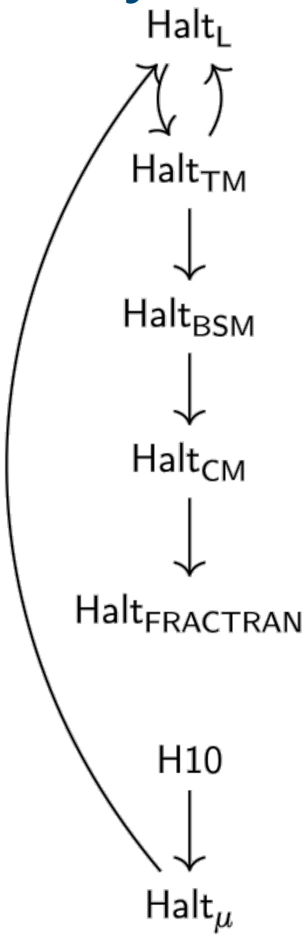
Lemma (Synthetic)

There is no enumerator for the complement of the halting problem, assuming CT.

Synthetic definition

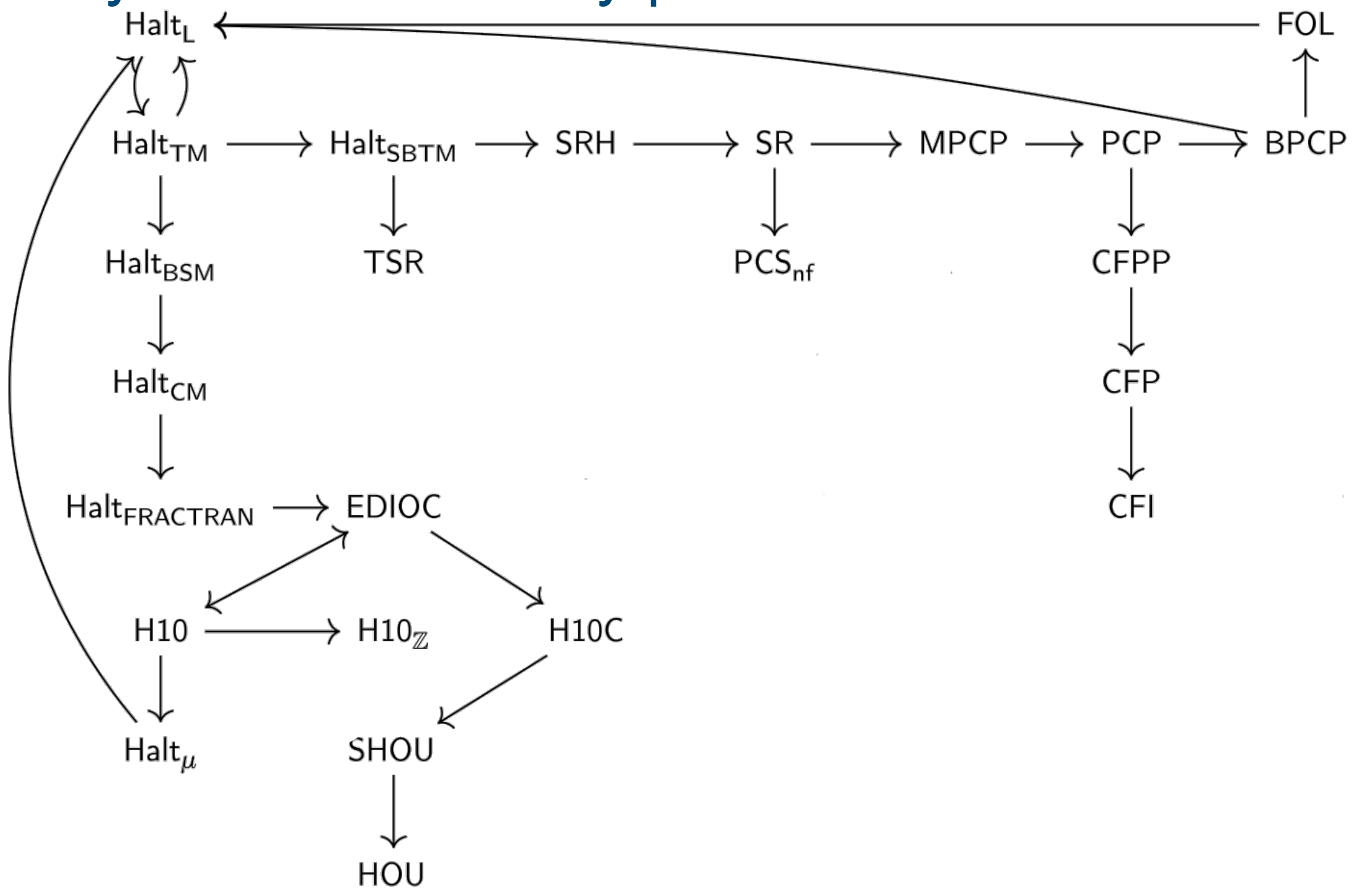
$$\mathcal{U}p := \mathcal{D}p \to \mathcal{E}(\overline{\mathsf{Halt}_{\mathsf{TM}}})$$

The Coq library of undecidability proofs



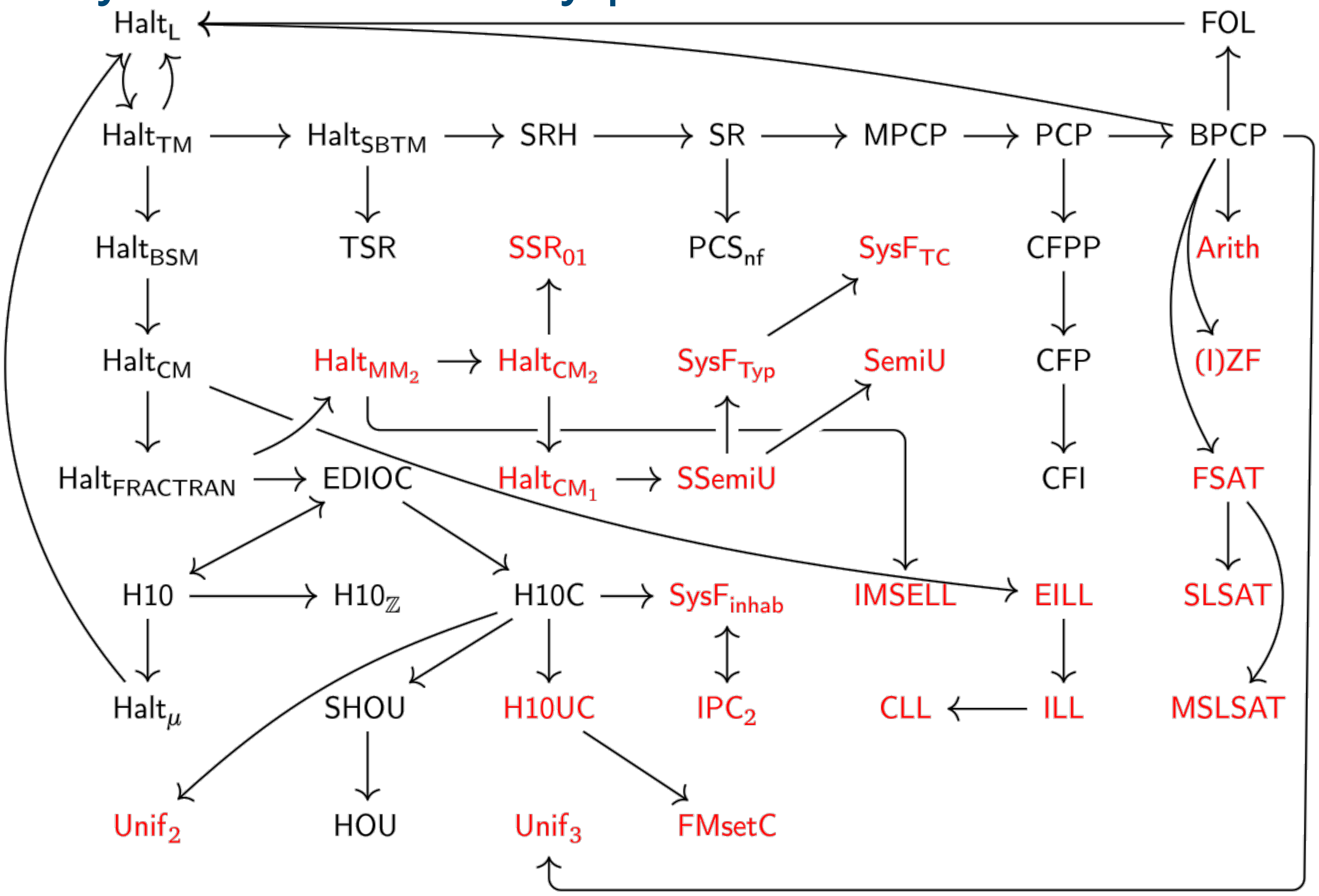
with Dominique Larchey-Wendling, Gert Smolka, Fabian Kunze, Max Wuttke …

The Coq library of undecidability proofs

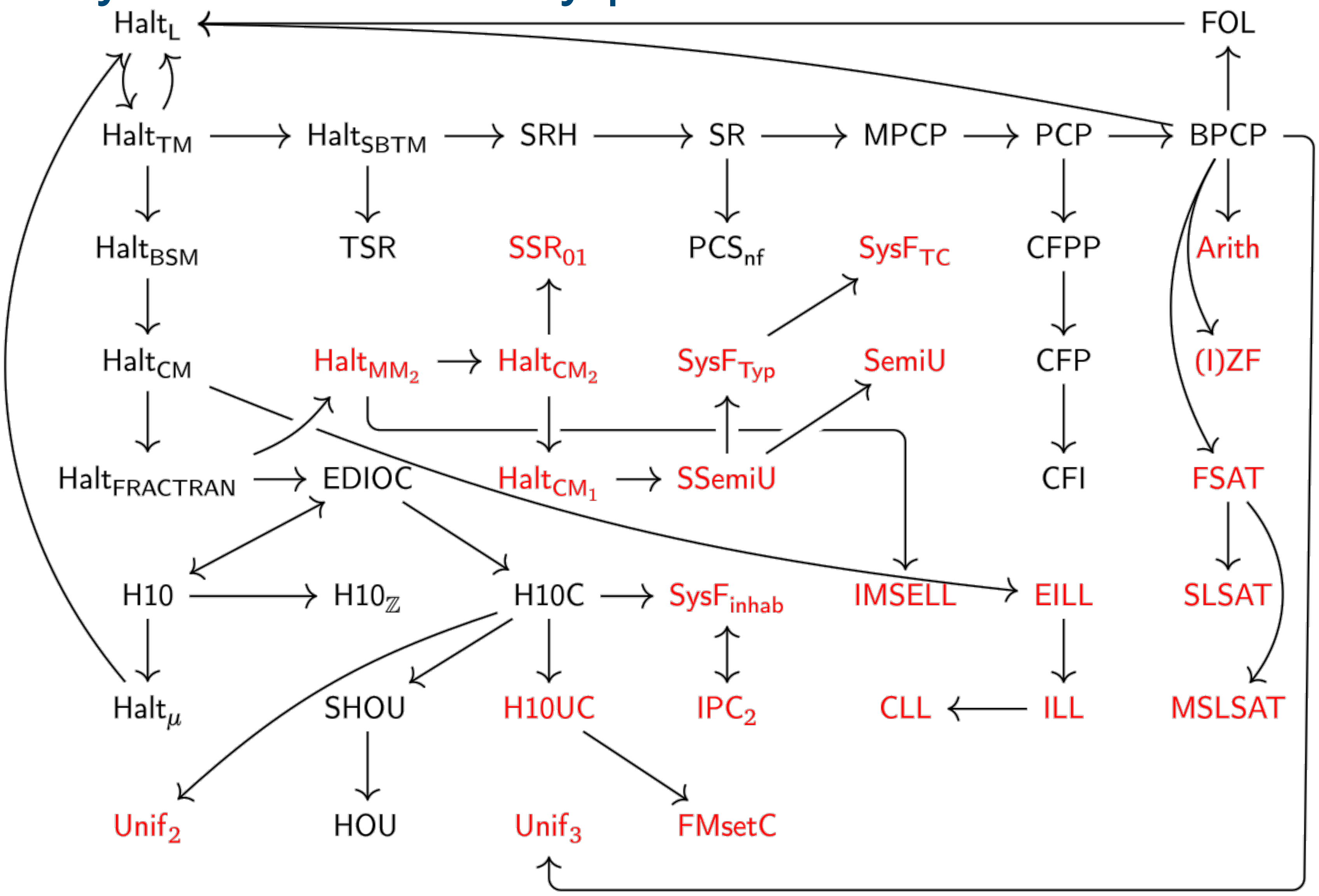


with … Edith Heiter, Dominik Kirst, Simon Spies, Dominik Wehr

The Coq library of undecidability proofs



The Coq library of undecidability proofs



117k lines of code, 12 contributers, larger than the `mathcomp` core library

Models of computation

- Equivalence proofs for computability of relations $\mathbb{N}^k \to \mathbb{N} \to \mathbb{P}$
- Identification of the weak call-by-value λ-calculus as sweet spot
  - extraction framework doing automatic computability proofs
  - can be used to prove many-one equivalence between problems
  - can be used to prove that SCT is a consequence of CT
  - even works as a foundation for complexity theory, see Fabian Kunze's work

Conclusion

- Machine-checked textbook proofs are feasible using synthetic approach, proofs can focus on mathematical essence.
- CIC allows these proofs to be classical and is an ideal ground for constructive reverse mathematics without choice.
- Lots of open questions regarding constructive status for even basic results.
- Machine-checked undecidability proofs from cutting-edge research are feasible, proofs can focus on inductive invariants.
- Avoid working in models of computation explicitly in a proof assistant, unless it is the weak call-by-value λ-calculus.

Conclusion

- Machine-checked textbook proofs are feasible using synthetic approach, proofs can focus on mathematical essence.
- CIC allows these proofs to be classical and is an ideal ground for constructive reverse mathematics without choice.
- Lots of open questions regarding constructive status for even basic results.
- Machine-checked undecidability proofs from cutting-edge research are feasible, proofs can focus on inductive invariants.
- Avoid working in models of computation explicitly in a proof assistant, unless it is the weak call-by-value λ-calculus.

Thank you!