The Rocq Library of Undecidability Proofs

Yannick Forster

INRIA, Cambium Team, Paris

EuroProofNet WG4

Work done over the last 8ish years

In this talk: work by Dominik Kirst, Gert Smolka, Dominique Larchey, Wendling, Andrej Dudenhefner.

The Rocq Undecidability Library has contributions by Dominique Larchey-Wendling, Andrej Dudenhefner, Janis Bailitis, Fabian Brenner, Edith Heiter, Marc Hermes, Johannes Hostert, Dominik Kirst, Mark Koch, Fabian Kunze, Gert Smolka, Simon Spies, Dominik Wehr, Maxi Wuttke, Nils Lauermann, Fabian Kunze, and Benjamin Peters.

How to formalise text books on computability theory?

How to formalise text books on computability theory?

Why is this area particularly hard to formalise?

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
   3.5 Kolmogorov complexity
   3.6 Kleene-Post and Post's hierarchy theorem
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 and Turing reducibility — relying on Church Turing thesis
   3.5 Kolmogorov complexity — relying on Church Turing thesis
   3.6 Kleene-Post and Post's hierarchy theorem — 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 and Turing reducibility
   3.5 Kolmogorov complexity
   3.6 Kleene-Post and Post's hierarchy theorem
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 and Turing reducibility
   3.5 Kolmogorov complexity
   3.6 Kleene-Post and Post's hierarchy theorem
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 and Turing reducibility
   3.5 Kolmogorov complexity
   3.6 Kleene-Post and Post's hierarchy theorem
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 and Turing reducibility
   3.5 Kolmogorov complexity
   3.6 Kleene-Post and Post's hierarchy theorem
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 and Turing reducibility
   3.5 Kolmogorov complexity
   3.6 Kleene-Post and Post's hierarchy theorem
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 and Turing reducibility
   3.5 Kolmogorov complexity
   3.6 Kleene-Post and Post's hierarchy theorem
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 and Turing reducibility
    3.5 Kolmogorov complexity
    3.6 Kleene-Post and Post's hierarchy theorem
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 and Turing reducibility ?
   3.5 Kolmogorov complexity ?
   3.6 Kleene-Post and Post's hierarchy theorem ?
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

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

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

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:

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

Then, with respect to W_y,

$$\begin{aligned} q_1(\overline{\mathfrak{x}^{(n)}}) &\to q_1 B(\overline{\mathfrak{x}^{(n)}}) \\ &\to q_2 BB(\overline{\mathfrak{x}^{(n)}}) \\ &\to \cdots \\ &\to q_{y+2}(\overline{y, \mathfrak{x}^{(n)}}). \end{aligned}$$

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 $q_{y+2}(\overline{y, \mathfrak{x}^{(n)}})$ that those of Z have on $q_1(\overline{y, \mathfrak{x}^{(n)}})$, we have

$$\Psi_{Z_y;A}^{(n)}(\mathfrak{x}^{(n)}) = \Psi_Z^{(1+n)}(y, \mathfrak{x}^{(n)}) = [r]_{1+n}^{A}(y, \mathfrak{x}^{(n)}). \qquad (1)$$

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),

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

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]

```
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

    list_encode (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)

    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 $[\exists^*\forall^2\exists^*, \mathsf{all}, (0)]$ 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 $[\exists^*\forall^2\exists^*, \mathsf{all}, (0)]$ fragment of FOL could be extended to include equality.

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

1984 Goldfarb is tasked to prove 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 is tasked to prove Gödel's claim, proves undecidability of this fragment.

Is there a need for machine-checked computability proofs?

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

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

1984 Goldfarb is tasked to prove Gödel's claim, proves undecidability of this fragment.

1988 Kfoury, Tiuryn, and Urzyczyn: decidability of semi-unification (POPL).

Is there a need for machine-checked computability proofs?

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

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

1984 Goldfarb is tasked to prove Gödel's claim, proves undecidability of this fragment.

1988 Kfoury, Tiuryn, and Urzyczyn: decidability of semi-unification (POPL).

1990 Kfoury, Tiuryn, and Urzyczyn: *un*decidability of semi-unification (LICS).

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 is tasked to prove Gödel's claim, proves undecidability of this fragment.

1988 Kfoury, Tiuryn, and Urzyczyn: decidability of semi-unification (POPL).

1990 Kfoury, Tiuryn, and Urzyczyn: *un*decidability of semi-unification (LICS).

1967 Minsky introduces 2-counter machines with inc and dec/jmp on zero, proves undecidability of 2CM-Halt with inc and dec/jmp on nonzero

Is there a need for machine-checked computability proofs?

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

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

1984 Goldfarb is tasked to prove Gödel's claim, proves undecidability of this fragment.

1988 Kfoury, Tiuryn, and Urzyczyn: decidability of semi-unification (POPL).

1990 Kfoury, Tiuryn, and Urzyczyn: *un*decidability of semi-unification (LICS).

1967 Minsky introduces 2-counter machines with inc and dec/jmp on zero, proves undecidability of 2CM-Halt with inc and dec/jmp on nonzero

2022 Dudenhefner proves decidability of 2CM-Halt with inc and dec/jmp on zero

State of the art in machine-checked proofs

Theory up to universal machines and Rice's theorem

2011 λ-calculus in HOL4 by Norrish

2017 weak call-by-value λ-calculus in Rocq by Forster and Smolka

2019 μ-recursive functions in Lean by Carneiro

2020 `PVS0` in `PVS` by Ferreira Ramos et al.

Miscellaneous results

2019 Bayer et al. prove Hilbert's 10th problem undecidable in Isabelle

2021 Kunze and Gäher prove Cook-Levin theorem in Rocq

2021 Forster, Kunze, Wuttke, Smolka formalise polynomial time equivalence of Turing machines and call-by-value λ-calculus

2023 Balbach proves Cook-Levin theorem in Isabelle

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

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:

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

Then, with respect to W_y,

$$\begin{aligned} q_1(\overline{\mathfrak{x}^{(n)}}) &\rightarrow q_1 B(\overline{\mathfrak{x}^{(n)}}) \\ &\rightarrow q_2 BB(\overline{\mathfrak{x}^{(n)}}) \\ &\rightarrow \cdots \\ &\rightarrow q_{y+2}(\overline{y, \mathfrak{x}^{(n)}}). \end{aligned}$$

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 $q_{y+2}(\overline{y, \mathfrak{x}^{(n)}})$ 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]

```
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
```

```
    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)
```

```
    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

*only possible in computational systems

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

*only possible in computational systems

Constructive mathematics to the rescue

Church-Turing thesis:

"Every intuitively computable function is μ-recursive."

Constructive mathematics to the rescue

Church-Turing thesis:
"Every intuitively computable function is μ-recursive."

Church's rule in computational systems:
whenever one can define a closed $f : \mathbb{N} \to \mathbb{N}$,
one could have actually defined a μ-recursive function computing f

$$\frac{\emptyset \vdash f : \mathbb{N} \to \mathbb{N}}{\emptyset \vdash \exists c : \mathbb{N}.\ \textit{the } c\textit{-th } \mu\textit{-recursive function computes } f}$$

Definitions

Analytic	**Synthetic**

Decidability

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

Definitions

Analytic	**Synthetic**
Decidability	
$\exists f : \mathbb{N} \to \mathbb{B}. \forall x.\ px \leftrightarrow fx = \text{true} \wedge f \textit{ is computable}$	$\exists f : \mathbb{N} \to \mathbb{B}. \forall x.\ px \leftrightarrow fx = \text{true}$
Semi-decidability	
$\exists f : \mathbb{N} \rightharpoonup \mathbb{N}. \forall x.\ px \leftrightarrow fx \downarrow \wedge f \textit{ is computable}$	$\exists f : \mathbb{N} \rightharpoonup \mathbb{N}. \forall x.\ px \leftrightarrow fx \downarrow$

Definitions

Analytic	**Synthetic**
Decidability	
$\exists f : \mathbb{N} \to \mathbb{B}. \forall x.\ px \leftrightarrow f x = \text{true}$ $\wedge f$ *is computable*	$\exists f : \mathbb{N} \to \mathbb{B}. \forall x.\ px \leftrightarrow f x = \text{true}$
Semi-decidability	
$\exists f : \mathbb{N} \rightharpoonup \mathbb{N}. \forall x.\ px \leftrightarrow f x \downarrow$ $\wedge f$ *is computable*	$\exists f : \mathbb{N} \rightharpoonup \mathbb{N}. \forall x.\ px \leftrightarrow f x \downarrow$
Many-one reducibility	
$\exists f : \mathbb{N} \to \mathbb{N}. \forall x.\ px \leftrightarrow q(f x)$ $\wedge f$ *is computable*	$\exists f : \mathbb{N} \to \mathbb{N}. \forall x.\ px \leftrightarrow q(f x)$

Definitions

Analytic	**Synthetic**
	Decidability
$\exists f : \mathbb{N} \to \mathbb{B}. \forall x.\ px \leftrightarrow fx = \text{true} \wedge f \textit{ is computable}$	$\exists f : \mathbb{N} \to \mathbb{B}. \forall x.\ px \leftrightarrow fx = \text{true}$
	Semi-decidability
$\exists f : \mathbb{N} \rightharpoonup \mathbb{N}. \forall x.\ px \leftrightarrow fx \downarrow \wedge f \textit{ is computable}$	$\exists f : \mathbb{N} \rightharpoonup \mathbb{N}. \forall x.\ px \leftrightarrow fx \downarrow$
	Many-one reducibility
$\exists f : \mathbb{N} \to \mathbb{N}. \forall x.\ px \leftrightarrow q(fx) \wedge f \textit{ is computable}$	$\exists f : \mathbb{N} \to \mathbb{N}. \forall x.\ px \leftrightarrow q(fx)$

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

Myhill isomorphism theorem

Theorem

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

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

jww Felix Jahn and Gert Smolka [CPP '23]

Church's rule is not true in classical systems...

$$\frac{\emptyset \vdash f : \mathbb{N} \to \mathbb{N}}{\emptyset \vdash \exists c : \mathbb{N}.\ \textit{the c-th } \mu\textit{-recursive function computes } f}$$

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

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

Troelstra and van Dalen [1988]

Church's rule is not true in classical systems...

$$\frac{\emptyset \vdash f : \mathbb{N} \to \mathbb{N}}{\emptyset \vdash \exists c : \mathbb{N}.\ \textit{the c-th } \mu\textit{-recursive function computes } f}$$

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

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

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]

Church's rule is not true in classical systems...

$$\frac{\emptyset \vdash f : \mathbb{N} \to \mathbb{N}}{\emptyset \vdash \exists c : \mathbb{N}.\ \textit{the } c\textit{-th } \mu\textit{-recursive function computes } f}$$

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

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

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, i.e. LEM)

Troelstra and van Dalen [1988]

Church's rule is not true in classical systems...

$$\frac{\emptyset \vdash f : \mathbb{N} \to \mathbb{N}}{\emptyset \vdash \exists c : \mathbb{N}.\ \textit{the } c\textit{-th } \mu\textit{-recursive function computes } f}$$

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

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

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, i.e. LEM)

Troelstra and van Dalen [1988]

CT is consistent in constructive systems

$$\mathsf{CT} := \forall f : \mathbb{N} \to \mathbb{N}.\textit{the } c\textit{-th } \mu\textit{-recursive function computes } f$$

- Heyting arithmetic, Kleene [1945]
- Russian style constructive mathematics, Markov [1954]
- In any system that has semantics via topoi, Hyland [1982]
- HoTT (MLTT + propositional truncation + univalence), Swan and Uemura [2019]
- MLTT, Pédrot [2024]

Slogans of (Rocq and Lean'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:

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

Slogans of (Rocq and Lean'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 **is not inconsistent in CIC**

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

CT **is not inconsistent in CIC**

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

decision can not be implemented

CT is not inconsistent in CIC

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

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

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

CT is not inconsistent in CIC

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

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

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

- ☑ G is functional
- ☐ G is total

CT **is not inconsistent in CIC**

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

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

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

- ☑ 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"

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

Relations to functions: Choice principles

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

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

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"

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

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"

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

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"

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

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"

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

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

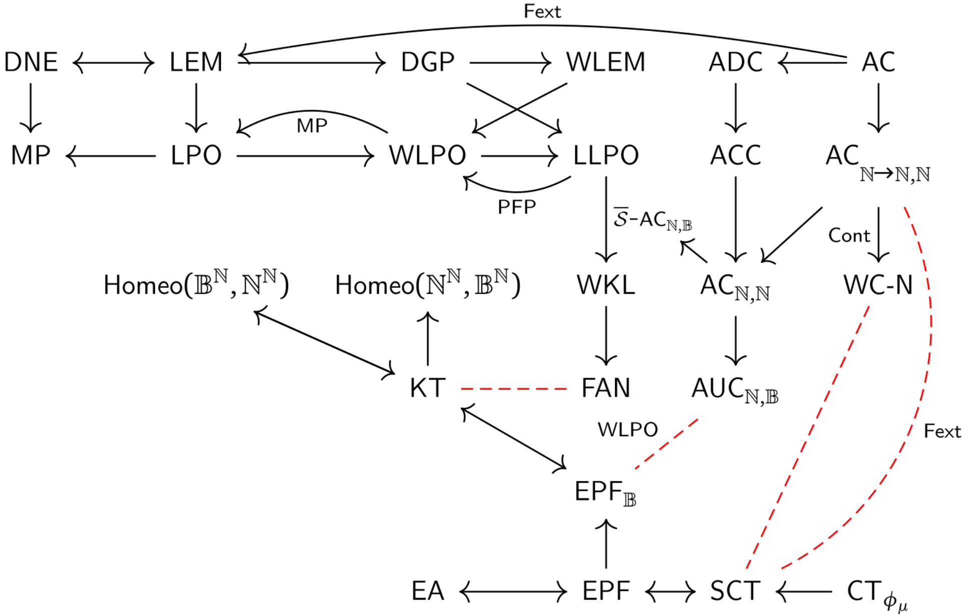
The axiom of choice: "every total relation contains a function"

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

Theorem

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

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


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
$Homeo(\mathbb{B}^{\mathbb{N}}, \mathbb{N}^{\mathbb{N}})$
$Homeo(\mathbb{N}^{\mathbb{N}}, \mathbb{B}^{\mathbb{N}})$
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}

Conjecture

The following are consistent in CIC:

- CT
- LEM
- functional extensionality, propositional extensionality (implies in particular PI)
- AC for relations: "Every total relation contains a total functional subrelation."

Textbook Computability Theory in Rocq

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 hierarchy theorem
4. Prove undecidability of concrete problems (PCP, CFGs)

Textbook Computability Theory in Rocq

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 hierarchy theorem ✓
4. Prove undecidability of concrete problems (PCP, CFGs, needs CT) ✓

The Rocq Library of Undecidability Proofs

Work done over the last 7ish years, mainly 2017-2022

The Rocq Undecidability Library has contributions by Dominique Larchey-Wendling, Andrej Dudenhefner, Dominik Kirst, Janis Bailitis, Fabian Brenner, Edith Heiter, Marc Hermes, Johannes Hostert, Mark Koch, Fabian Kunze, Gert Smolka, Simon Spies, Dominik Wehr, Maxi Wuttke, Niklas Mück, Haoyi Zeng, Nils Lauermann, and Benjamin Peters.

Synthetic undecidability via reducibility

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)

If $\mathsf{Halt}_{\mathsf{TM}} \preceq_m p$*, then* p *is not decidable.*

Synthetic undecidability via reducibility

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.

Theorem (Synthetic)

If $\mathsf{Halt}_{\mathsf{TM}} \preceq_m p$*, then* p *is not decidable, assuming* CT.

Synthetic undecidability via reducibility

Analytic definition

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

Lemma (Synthetic)

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

Theorem (Synthetic)

If $\mathsf{Halt}_{\mathsf{TM}} \preceq_m p$*, then* p *is not decidable, assuming* CT.

Synthetic definition

$$\mathcal{U} p := \overline{\mathsf{Halt}_{\mathsf{TM}}} \preceq_m p$$

Synthetic undecidability via reducibility in different systems

- in Rocq: $\mathcal{U}$ can be defined in $\mathbb{P}$, proofs can be classical

Synthetic undecidability via reducibility in different systems

- in Rocq: $\mathcal{U}$ can be defined in $\mathbb{P}$, proofs can be classical
  e.g.: subsingleton sets are always decidable

Synthetic undecidability via reducibility in different systems

- in Rocq: $\mathcal{U}$ can be defined in $\mathbb{P}$, proofs can be classical
  e.g.: subsingleton sets are always decidable
- in Lean: $\mathcal{U}$ must be defined in $\mathbb{T}$, only verification can be classical

Synthetic undecidability via reducibility in different systems

- in Rocq: $\mathcal{U}$ can be defined in $\mathbb{P}$, proofs can be classical
  e.g.: subsingleton sets are always decidable
- in Lean: $\mathcal{U}$ must be defined in $\mathbb{T}$, only verification can be classical
  $\Rightarrow$ need to manually check axiom-freeness of reduction

Synthetic undecidability via reducibility in different systems

- in Rocq: $\mathcal{U}$ can be defined in $\mathbb{P}$, proofs can be classical
  e.g.: subsingleton sets are always decidable
- in Lean: $\mathcal{U}$ must be defined in $\mathbb{T}$, only verification can be classical
  $\Rightarrow$ need to manually check axiom-freeness of reduction
  $\Rightarrow$ strange theorems: countable unions of r.e. sets are r.e.

Why does this all work?

Rocq and Lean are programming languages, proofs are not supposed to be executed.

Why does this all work?

Rocq and Lean are programming languages, proofs are not supposed to be executed.

UNIVERSITE PARIS VII

THESE DE DOCTORAT

Spécialité : **INFORMATIQUE**

Présentée par **Christine PAULIN-MOHRING**

Sujet :

Extraction de programmes
dans le Calcul des Constructions

Soutenue le 27 janvier 1989 devant la commission d'examen :

Jean-Louis Krivine	Président et rapporteur
Thierry Coquand	Rapporteur
Guy Cousineau	Examinateurs
Gérard Huet	
Per Martin-Löf	
Michel Sintzoff	

N° d'ordre : 7567

Thèse de doctorat

présentée à

L'Université de Paris-Sud
U.F.R. Scientifique d'Orsay

par

Pierre Letouzey

pour obtenir

le grade de docteur en sciences
de l'Université de Paris XI Orsay
spécialité : Informatique

Sujet :

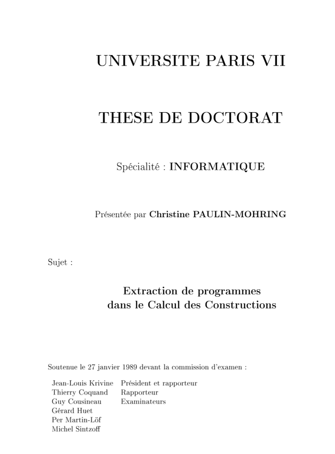
Programmation fonctionnelle certifiée
L'extraction de programmes dans l'assistant Coq

Soutenue le 9 juillet 2004 devant la commission d'examen composée de

M.	Leroy Xavier	président
M.	Berardi Stefano	rapporteurs
M.	Monin Jean-François	
Mme	Benzaken Véronique	examinateurs
M.	Schwichtenberg Helmut	
Mme	Paulin Christine	directeur

Why does this all work?

Rocq and Lean are programming languages, proofs are not supposed to be executed.



UNIVERSITE PARIS VII

THESE DE DOCTORAT

Spécialité : **INFORMATIQUE**

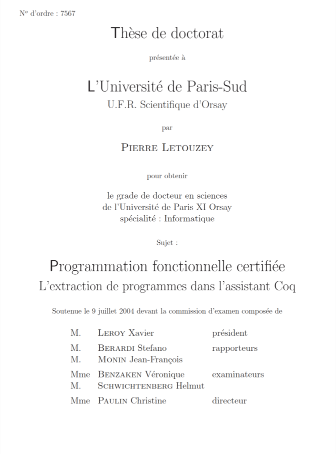
Présentée par **Christine PAULIN-MOHRING**

Sujet :

Extraction de programmes
dans le Calcul des Constructions

Soutenue le 27 janvier 1989 devant la commission d'examen :

Jean-Louis Krivine	Président et rapporteur
Thierry Coquand	Rapporteur
Guy Cousineau	Examinateurs
Gérard Huet	
Per Martin-Löf	
Michel Sintzoff	



N° d'ordre : 7567

Thèse de doctorat

présentée à

L'Université de Paris-Sud
U.F.R. Scientifique d'Orsay

par

PIERRE LETOUZEY

pour obtenir

le grade de docteur en sciences
de l'Université de Paris XI Orsay
spécialité : Informatique

Sujet :

Programmation fonctionnelle certifiée
L'extraction de programmes dans l'assistant Coq

Soutenue le 9 juillet 2004 devant la commission d'examen composée de

M.	LEROY Xavier	président
M.	BERARDI Stefano	rapporteurs
M.	MONIN Jean-François	
Mme	BENZAKEN Véronique	examinateurs
M.	SCHWICHTENBERG Helmut	
Mme	PAULIN Christine	directeur

(Total) programming languages remain systems for constructive mathematics!

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 an undecidable p, there is a μ-recursive 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 an undecidable 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 an undecidable 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

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

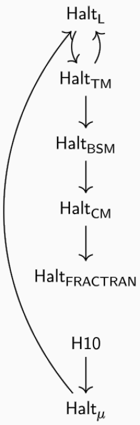
Lemma (Synthetic)

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

Synthetic definition

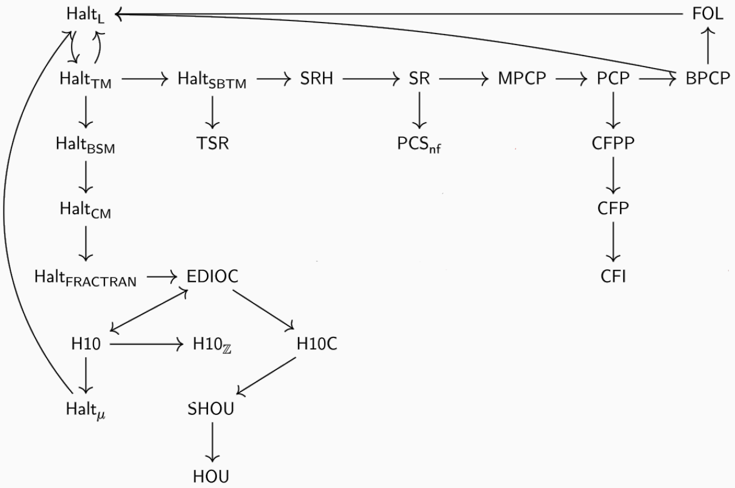
$$\mathcal{U}p := \mathcal{D}p \to \mathcal{E}(\overline{\mathsf{Halt}_{\mathsf{TM}}})$$

The Rocq library of undecidability proofs



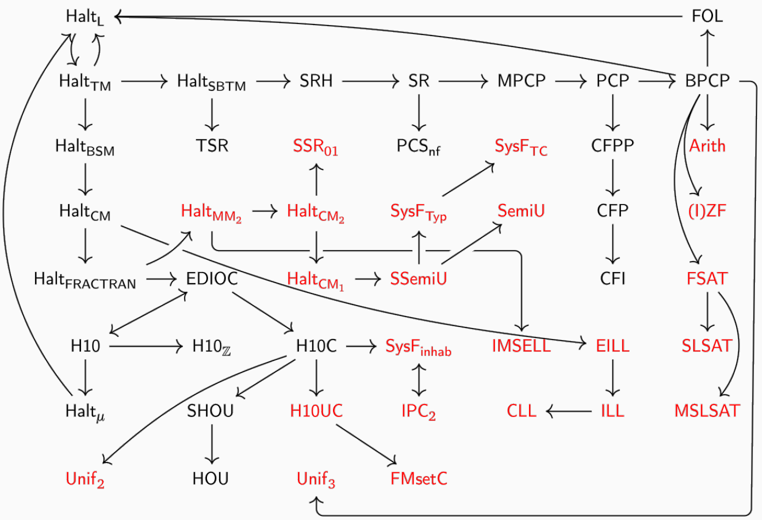
with Dominique Larchey-Wendling, Gert Smolka, Fabian Kunze, Max Wuttke ...

The Rocq library of undecidability proofs

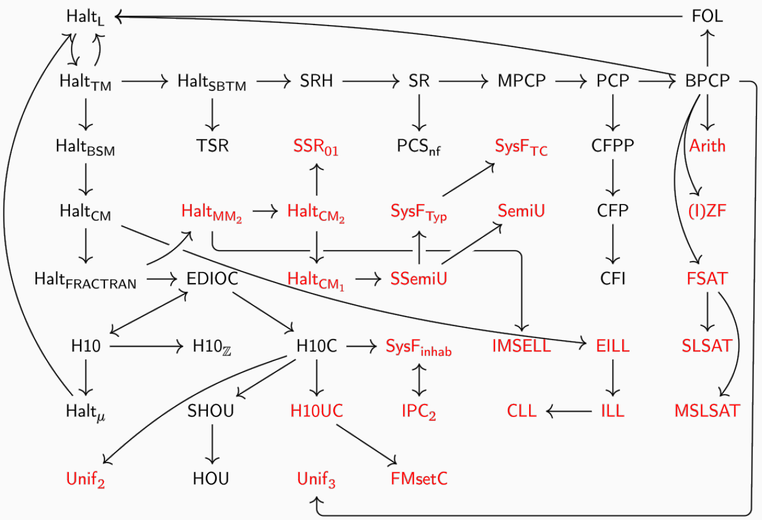


with ... Edith Heiter, Dominik Kirst, Simon Spies, Dominik Wehr

The Rocq library of undecidability proofs



The Rocq library of undecidability proofs



$\sim$100k lines of code, 14 contributers

Seed problems

- Halting problem for single-tape two-symbol Turing machines
- Post correspondence problem (PCP)
- Halting problem for two counters machines due to Minsky
- Halting problem for FRACTRAN programs due to Conway
- Satisfiability of elementary Diophantine constraints of the form $x = 1, x = y + z$ or $x = y \cdot z$
- Halting problem for one counter machines
- Solvability of finite multiset constraint
- Simple semi-Thue system 01 rewriting

Decidable problems

- Two-counter Minsky Program Machine Halting. The definition follows exactly Minsky's book (Chapter 11, Table 11.1-1), and is different from two counter machines.
- Reversible Two-counter Machine Halting.
- Two-counter Machine Uniform Mortality.
- Two-counter Machine Uniform Boundedness.
- First-order unification

Hardest problems

- Hilbert's tenth problem (Davis, Putnam, Robinson, Matiyasevich)
- Higher-order unification (Huet, Goldfarb, Dowek)
- Semi-unification (Kfoury, Tiuryn, and Urzyczyn)
- Subtyping and type checking of System F (Wells)
- Lambda definability (Loader)

Forster and Larchey-Wendling [2018], Forster and Spies [2019], Dudenhefner [2020, 2021, 2024]

First-order library, modal logic

- Undecidability and Trakthenbrot's theorem
- Completeness theorems and constructive analysis
- Shortest incompleteness proof ever, relying on synthetic computability
- Tennenbaum's theorem
- Löb's theorem
- Löwenheim Skolem
- Bi-intuitionistic logic, (propositional) modal logics

Dominik Kirst with Forster, Smolka [2019], Larchey-Wendling [2020], Forster, Wehr [2020], Hagemeier [2022], Hermes [2022], Hostert, Dudenhefner [2022], Shillito [2024], Bailitis, Forster [2025], Zeng [2025].

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
  - ad-hoc extraction framework doing automatic computability proofs
  - can be used to prove many-one equivalence between problems
  - even works as a foundation for (analytic) complexity theory, see Fabian Kunze's work

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
  - ad-hoc extraction framework doing automatic computability proofs
  - can be used to prove many-one equivalence between problems
  - even works as a foundation for (analytic) complexity theory, see Fabian Kunze's work
- **A good framework for automatic computability proofs is missing in any PA.**

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
    - ad-hoc extraction framework doing automatic computability proofs
    - can be used to prove many-one equivalence between problems
    - even works as a foundation for (analytic) complexity theory, see Fabian Kunze's work
- **A good framework for automatic computability proofs is missing in any PA.**
  $\Rightarrow$ We do not have good enough meta-programming in any PA

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
  - ad-hoc extraction framework doing automatic computability proofs
  - can be used to prove many-one equivalence between problems
  - even works as a foundation for (analytic) complexity theory, see Fabian Kunze's work
- **A good framework for automatic computability proofs is missing in any PA.**

  $\Rightarrow$ We do not have good enough meta-programming in any PA (yet)

CI, PRs, etc

- one branch per Rocq versions, new results only go to newest branch
- a `main` branch where Rocq devs used to send commits to keep Rocq's CI working
- not in the Rocq CI anymore because of new rules
- bad code quality: we valued engagement and growth more than maintainability – but still never had to drop code

A typical thesis project at Saarland University

- Split into 270h “seminar” phase and 3 months thesis phase
- Weekly 1h meetings, in crunch phase maybe more
- No Rocq code in meetings, ever. Practice to condense problems for advisors.
- First talk: after 90h, explanation of the problem, 15min
- Second talk: after 260h, recap and goals of the project, 15min
- Final talk: Mimicking conference talk, 20min
- Accompanied by ∼60 page thesis
- Offer: we co-write paper, or we write paper for you, publish at ITP, CPP, FSCD

Conclusion

- Computability theory proofs have high amount of invisible math
- Lean, Rocq, Agda allow keeping this invisible for computability due to computational / constructive foundations
- Undecidability library covers almost all basic problems, running out of good student projects
- Self-contained elementary formalisation projects are ideal intro to research
- We know how to do computability theory, even with oracles, based on axioms or automation
- Open problem: How to do complexity theory?

Conclusion

- Computability theory proofs have high amount of invisible math
- Lean, Rocq, Agda allow keeping this invisible for computability due to computational / constructive foundations
- Undecidability library covers almost all basic problems, running out of good student projects
- Self-contained elementary formalisation projects are ideal intro to research
- We know how to do computability theory, even with oracles, based on axioms or automation
- Open problem: How to do complexity theory?

Thank you!