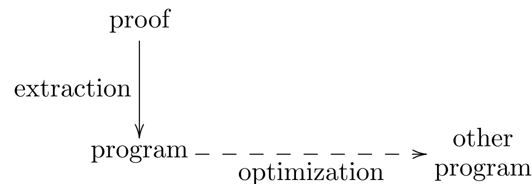
A Proof-Theoretic Account of Primitive Recursion and Primitive Iteration

LUCA CHIARABINI
Università degli Studi di Parma
Viale G.P.Usberti, 53/A - 43124 Parma , Italy
luca.chiarabini@unipr.it
and
OLIVIER DANVY
Department of Computer Science, Aarhus University
Aabogade 34, DK-8200 Aarhus N, Denmark
danvy@cs.au.dk

We revisit both the usual "going-up" induction principle and Manna and Waldinger's "going-down" induction principle for primitive recursion, à la Gödel, and primitive iteration, à la Church. We use 'Kleene's trick' to show that primitive recursion and primitive iteration are as expressive as each other, even in the presence of accumulators. As a result, we can directly extract a variety of recursive and iterative functional programs of the kind usually written or optimized by hand.

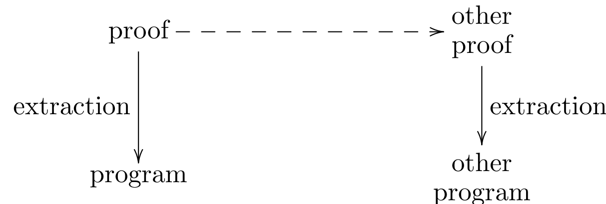
1. INTRODUCTION

There is proof theory, and there is programming practice. In theory, correct programs are extracted from proofs, and in practice, the extracted programs need to be optimized to run efficiently:



For example, for one recursive definition of the factorial function, there is an array of iterative versions, and the literature is replete with clever ways to go from the recursive definition to one of the iterative versions [8, 10, 13].

Our goal here is to give a proof-theoretic account of this cleverness so that the programs that run are the programs that are extracted:



We focus on primitive recursion, à la Gödel, and primitive iteration, à la Church.

Recursion over $\mathbf{N}$	Recursion over $\mathbf{L}(\rho)$
$\mathcal{R}_{\mathbf{N}}^{\sigma} : \sigma \to (\mathbf{N} \to \sigma \to \sigma) \to \mathbf{N} \to \sigma$	$\mathcal{R}_{\mathbf{L}(\sigma_1)}^{\sigma_2} : \sigma_2 \to (\sigma_1 \to \mathbf{L}(\sigma_1) \to \sigma_2 \to \sigma_2) \to \mathbf{L}(\sigma_1) \to \sigma_2$
$\mathcal{R}_{\mathbf{N}}^{\sigma}\, b\, f\, 0 \longmapsto b$	$\mathcal{R}_{\mathbf{L}(\sigma_2)}^{\sigma_1}\, b\, f\, \mathsf{nil} \longmapsto b$
$\mathcal{R}_{\mathbf{N}}^{\sigma}\, b\, f\, (\mathsf{Succ}\, n) \longmapsto f\, n\, (\mathcal{R}_{\mathbf{N}}^{\sigma}\, b\, f\, n)$	$\mathcal{R}_{\mathbf{L}(\rho)}^{\sigma}\, b\, f\, (a :: l) \longmapsto f\, a\, (\mathcal{R}_{\mathbf{L}(\rho)}^{\sigma}\, b\, f\, l)$

Recursion over $\mathbf{B}$	Projections	Application
$\mathcal{R}_{\mathbf{B}}^{\sigma} : \sigma \to \sigma \to \mathbf{B} \to \sigma$	$\pi_0(r, s) \longrightarrow r$	$(\lambda x.r)s \longrightarrow r[x := s]$
$(\mathcal{R}_{\mathbf{B}}^{\sigma}\, r\, s)\mathsf{tt} \longmapsto r$	$\pi_1(r, s) \longrightarrow s$	
$(\mathcal{R}_{\mathbf{B}}^{\sigma}\, r\, s)\mathsf{ff} \longmapsto s$		

Fig. 1. Conversion rules for Gödel's System T

Overview. We first make a short excursus about extracting programs from proofs (Section 2) with Minlog,[1] the proof assistant we used to develop all the proofs presented through the paper. We then review primitive up/down recursion and iteration (Section 3) and we study their expressive power (Section 4). Next, we turn to their accumulator-based counterparts (Section 5). Our running example, in Section 3, Section 4, and Section 5, is the factorial function. In Section 6, we extract three iterative versions of this function with an accumulator.

2. MODIFIED REALIZABILITY FOR FIRST-ORDER MINIMAL LOGIC

2.1 Gödel's System T

Types are built from base types and from compound types. Base types are natural numbers ($\mathbf{N}$) and booleans ($\mathbf{B}$). Compound types are lists with elements of some type ($\mathbf{L}(\sigma)$), functions ($\to$), and pairs ($\times$). The *terms* of Gödel's System T [4] are the terms of the simply typed λ-calculus with pairs, projections, and constants (constructors and recursive operators over natural numbers, booleans, and lists):

$$\begin{aligned} \mathit{Type} \ni \sigma, \sigma_1, \sigma_2 &::= \mathbf{N} \mid \mathbf{B} \mid \mathbf{L}(\sigma) \mid \sigma_1 \to \sigma_2 \mid \sigma_1 \times \sigma_2 \\ \mathit{Const} \ni c &::= 0 \mid \mathsf{Succ} \mid \mathsf{tt} \mid \mathsf{ff} \mid \mathsf{nil} \mid \mathsf{cons} \mid \mathcal{R}_{\mathbf{N}}^{\sigma} \mid \mathcal{R}_{\mathbf{B}}^{\sigma} \mid \mathcal{R}_{\mathbf{L}(\sigma_1)}^{\sigma_2} \\ \mathit{Terms} \ni t, t_0, t_1, t_2 &::= c \mid x^{\sigma} \mid (\lambda x^{\sigma_1}.t^{\sigma_2})^{\sigma_1 \to \sigma_2} \mid ({t_0}^{\sigma_1 \to \sigma_2}\, {t_1}^{\sigma_1})^{\sigma_2} \mid \\ &\quad\;\; ({t_1}^{\sigma_1}, {t_2}^{\sigma_2})^{\sigma_1 \times \sigma_2} \mid (\pi_1\, t^{\sigma_1 \times \sigma_2})^{\sigma_1} \mid (\pi_2\, t^{\sigma_1 \times \sigma_2})^{\sigma_2} \end{aligned}$$

As an abbreviation, $t_0 :: \cdots :: t_n :: \mathsf{nil}$ represents a list with $n + 1$ elements.

We equip this calculus with the usual conversion rules for the recursive operators, applications and projections (Figure 1).

2.2 Heyting Arithmetic

We define Heyting Arithmetic HA^{ω} [10, Page 240] for our language based on Gödel's System T, which is finitely typed. We define *negation* $\neg\varphi$ by $\varphi \to \bot$.

Formulas: Atomic formulas ($\mathrm{P}\vec{t}\vec{\rho}$) (P a predicate symbol, $\vec{t}$, $\vec{\rho}$ lists of terms and types), $\varphi \to \psi$, $\forall x^{\rho}\varphi$, $\exists x^{\rho}\varphi$, $\varphi \wedge \psi$.

[1] http://www.minlog-system.de

Derivations	Terms	Open assumptions
$\dfrac{}{\varphi}$ (ASS)	u^{φ}	$\{u\}$
$\dfrac{\begin{matrix}\vert M & \vert N \\ \varphi & \psi\end{matrix}}{\varphi \wedge \psi}$ $(\wedge^{+})$	$(\langle M^{\varphi}, N^{\psi}\rangle^{\varphi\wedge\psi})$	$\mathrm{OA}(M) \cup \mathrm{OA}(N)$
$\dfrac{\begin{matrix}\vert M \\ \varphi \wedge \psi\end{matrix}}{\varphi}$ $(\wedge^{-}_{0})$	$(M^{\varphi\wedge\psi}0)^{\varphi}$	$\mathrm{OA}(M)$
$\dfrac{\begin{matrix}\vert M \\ \varphi \wedge \psi\end{matrix}}{\psi}$ $(\wedge^{-}_{1})$	$(N^{\varphi\wedge\psi}1)^{\psi}$	$\mathrm{OA}(N)$
$\dfrac{\begin{matrix}[u : \varphi] \\ \vert M \\ \psi\end{matrix}}{\varphi \to \psi}$ $(\to^{+}_{u})$	$(\lambda u^{\varphi} M^{\psi})^{\varphi\to\psi}$	$\mathrm{OA}(M)\backslash\{u\}$
$\dfrac{\begin{matrix}\vert M & \vert N \\ \varphi \to \psi & \varphi\end{matrix}}{\psi}$ $(\to^{-})$	$(M^{\varphi\to\psi} N^{\varphi})^{\psi}$	$\mathrm{OA}(M) \cup \mathrm{OA}(N)$
$\dfrac{\begin{matrix}\vert M \\ \varphi\end{matrix}}{\forall x^{\rho}\varphi}$ $(\forall^{+})$ with Var. cond.	$(\lambda x^{\rho} M^{\varphi})^{\forall x^{\rho}\varphi}$	$\mathrm{OA}(M)$
$\dfrac{\begin{matrix}\vert M & \\ \forall x^{\rho}\varphi(x) & t^{\rho}\end{matrix}}{\varphi(t)}$ $(\forall^{-})$	$(M^{\forall x^{\rho}\varphi(x)} t^{\rho})^{\varphi}$	$\mathrm{OA}(M)$
$\dfrac{\begin{matrix}\vert M \\ \varphi\end{matrix}}{\forall^{\mathsf{nc}} x^{\rho}\varphi}$ $(\forall^{\mathsf{nc}+})$ with Var. cond. and $x \notin [\![M]\!]$	$(\lambda^{\mathsf{nc}} x^{\rho} M^{\varphi})^{\forall^{\mathsf{nc}} x^{\rho}\varphi}$	$\mathrm{OA}(M)$
$\dfrac{\begin{matrix}\vert M & \\ \forall^{\mathsf{nc}} x^{\rho}\varphi(x) & t^{\rho}\end{matrix}}{\varphi(t)}$ $(\forall^{\mathsf{nc}-})$	$(M^{\forall^{\mathsf{nc}} x^{\rho}\varphi} t^{\rho})^{\varphi}$	$\mathrm{OA}(M)$

"with Var. cond." stands for "provided $x^{\rho} \notin \mathrm{FV}(\varphi)$, for any $u^{\varphi} \in \mathrm{OA}(M)$"

Fig. 2. Derivation terms for $\forall$, $\forall^{\mathsf{nc}}$, $\to$ and $\wedge$

Derivations: By the Curry-Howard correspondence [10] it is convenient to write derivations as terms: we define λ-terms M^φ for natural deduction proofs of formulas φ together with the set $\mathrm{OA}(M)$ of *o*pen *a*ssumptions in M (see Figure 2). Usually we will omit type and formula indices in derivations if they are uniquely determined by the context or if they are not relevant.

We will use two special quantifiers $\forall^{\mathsf{nc}}/\exists^{\mathsf{nc}}$ (introduction and elimination rules in Figure 2) to indicate that there should be **n**o **c**omputational **c**ontent [11]. The logical meaning of the universal quantifiers is unchanged. However, we have to observe a special *variable condition* for $\forall^{\mathsf{nc}+}$: the variable to be abstracted should not be a *computational variable* in the given proof, i.e., the extracted program of this proof should not depend on x.

We use $\exists$ and $\vee$ in our logic, if we allow appropriate axioms as constant derivation terms:

$$\begin{aligned} \exists^+_{x^\rho,\varphi} &: \forall x^\rho(\varphi \to \exists x^\rho \varphi) \\ \exists^-_{x^\rho,\varphi,\psi} &: \exists x^\rho \varphi \to (\forall x^\rho \varphi \to \psi) \to \psi \text{ with } x \notin FV(\psi) \end{aligned}$$

We can define $\vee$ from $\exists$ via:

$$\varphi \vee \psi := \exists p^{\mathbf{B}}(p \to \varphi) \wedge ((p \to \bot) \to \psi)$$

We use the following axioms to perform proofs by induction over naturals ($\mathbf{N}$), booleans ($\mathbf{B}$) and lists of elements of type ρ ($\mathbf{L}(\rho)$):

$$\begin{aligned} \mathsf{Ind}_{n,A} &: A[n \mapsto 0] \to \forall n(A \to A[n \mapsto \mathrm{S}n]) \to \forall n^{\mathbf{N}} A, \\ \mathsf{Ind}_{p,A} &: A[p \mapsto \mathsf{tt}] \to A[p \mapsto \mathsf{ff}] \to \forall p^{\mathbf{B}} A, \\ \mathsf{Ind}_{l,A} &: A[l \mapsto \mathsf{nil}] \to (\forall x, l.A \to A[l \mapsto x :: l]) \to \forall l^{\mathbf{L}(\alpha)} A \end{aligned}$$

Finally we use the constant derivation term (IF_φ),

$$\mathsf{IF}_\varphi : \forall p^{\mathbf{B}}(p \to \varphi) \to ((p \to \bot) \to \varphi) \to \varphi$$

to perform *case distinction* on boolean terms w.r.t. a goal formula φ.

2.3 Short Excursus in Program Extraction from Proofs

By definition, existence proofs have computational content: we can exhibit a witness, which thus contributes to the extracted program. Realizability extends this idea to other formulas; it can be seen as an incarnation of the Brouwer-Heyting-Kolmogorov interpretation [12] of proofs.

2.3.1 *Type of a Formula.* We note $\tau(\varphi)$ the type of the term (or "program") to be extracted from a proof of φ. More precisely, to every formula φ it is possible to assign an object of type $\tau(\varphi)$ (a type or the "nulltype" symbol ε). In case $\tau(\varphi) = \varepsilon$ proofs of φ have no computational content; such formulas φ are called *Harrop formulas*. (See Appendix A.1.)

2.3.2 *Extraction Map.* From every derivation M of a computationally meaningful formula φ (that is, $\tau(\varphi) \neq \varepsilon$), it is possible to define its *extracted program* $[\![M]\!]$ of type $\tau(\varphi)$ [1, 7, 9, 10]. If $\tau(\varphi) = \varepsilon$ then $[\![M]\!] = \varepsilon$: the notational overlap does not hurt, as those programs are dropped out anyway. (See Appendix A.2.)

2.3.3 *Realization of a formula.* The correctness of the extracted programs is guaranteed by the notion of *modified realizability.* Intuitively, if t is the extracted program from the derivation M of the formula $\forall x \exists y. P(x, y)$ (φ), then for each x the formula $P(x, t(x))$ is provably correct (*Soundness*), i.e., t *realizes* φ (according to modified realizability), written $(t\ \boldsymbol{mr}\ \varphi)$. (See Appendix A.3.)

Theorem (Soundness). *Let M be a derivation of a formula φ from assumptions $u_i : \varphi_i$. Then we can find a derivation of the formula $([\![M]\!]\ \boldsymbol{mr}\ \varphi)$ from assumptions $\bar{u}_i : x_{u_i}\ \boldsymbol{mr}\ \varphi_i$.*

Proof. By structural induction on M. □

In the following, for readability, all the extracted programs will be displayed in a the syntax of Standard ML.

3. UP AND DOWN INDUCTION PRINCIPLES OVER NATURAL NUMBERS

We successively review primitive recursion, i.e., 'rec' in Gödel's System T (Section 3.1) and then primitive iteration, i.e., Church numerals (Section 3.2). We then turn to symmetric analogues that correspond to Manna and Waldinger's "going down" recursion (Section 3.3) and induction (Section 3.4). In reference to Manna and Waldinger's pioneering work [8], we refer to these induction principles as 'up' and 'down.'

3.1 Up primitive recursive induction

Here is the proof principle for primitive recursion:

$$\frac{\begin{array}{c}|Z\\P(0)\end{array} \qquad \begin{array}{c}|S\\\forall n(P(n) \to P(n+1))\end{array}}{\forall n P(n)}\ (\text{up-prim-rec})$$

Manna and Waldinger refer to it as 'going up' since $P(n)$ is needed to deduce $P(n + 1)$. The corresponding synthesized functional `Up.prim_rec` is displayed in Figure 3. There, `z` is extracted from $[\![Z]\!]$ and `s` from $[\![S]\!]$. The computation is driven by the natural number denoted by the input variable `n`: computing the result for n requires the result for $n-1$ to be computed, until the base case $n = 0$ is reached in a trail of nested applications of the function denoted by `s`. The computation then proceeds going up from 0 to the given natural number.

The traditional definition of the factorial function is a straightforward example of primitive recursion. It is obtained as an instance of `Up.prim_rec` where `z` is instantiated with the neutral element for multiplication (`1`) and `s` with the (curried) multiplication function (`fn i => fn c => (i + 1) * c`):

```
fun fact n
    = Up.prim_rec (1, fn i => fn c => (i + 1) * c) n
```

The extracted program reads as follows:

```
fun up_prim_rec_fact n
    = let fun visit m
              = if m = 0 then 1 else m * (visit (m - 1))
      in visit n
      end
```

```
structure Up
= struct
    (*  prim_rec : 'a * (int -> 'a -> 'a) -> int -> 'a  *)
    fun prim_rec (z, s) n
        = let fun visit m
                  = if m = 0 then z else s (m - 1) (visit (m - 1))
          in visit n
          end

    (*  prim_iter : 'a * ('a -> 'a) -> int -> 'a  *)
    fun prim_iter (z, s) n
        = let fun visit m
                  = if m = 0 then z else s (visit (m - 1))
          in visit n
          end
  end
```

Fig. 3. Synthesized functionals for up induction

3.2 Up primitive iterative induction

Here is the proof principle for primitive iteration:

$$\dfrac{\begin{matrix} |Z \\ P(0) \end{matrix} \qquad \begin{matrix} |S \\ \forall^{\mathsf{nc}} n (P(n) \to P(n+1)) \end{matrix}}{\forall n P(n)} \;(\text{up-prim-iter})$$

The difference between primitive and iterative induction is that in the iterative case, we quantify non computationally over n in the inductive step. One can then synthesize the functional for up primitive iteration `Up.prim_rec` in Figure 3. Again, there, `z` is extracted from $[\![Z]\!]$ and `s` from $[\![S]\!]$.

To define the factorial function as an instance of `Up.prim_iter`, we must generalize Kleene's trick to compute the predecessor function over Church numerals [6]. We instantiate `z` with `(1, 1)` and `s` with `fn (i, c) => (i + 1, i * c)` in `Up.prim_iter`:

```
fun fact n
    = let val (i, c) = Up.prim_iter ((1, 1), fn (i, c) => (i + 1, i * c)) n
      in c
      end
```

The extracted program reads as follows:

```
fun up_prim_iter_fact n
    = let fun visit m
            = if m = 0 then (1, 1) else let val (i, c) = visit (m - 1)
                                        in (i + 1, i * c)
                                        end
      in let val (i, c) = visit n in c end
      end
```

3.3 Down primitive recursive induction

Manna and Waldlinger also present a 'going down' version of primitive recursion:

$$\dfrac{\begin{matrix}|Z \\ Q(n)\end{matrix} \qquad \begin{matrix}|S \\ \forall m(Q(m+1) \to Q(m))\end{matrix}}{Q(0)} \ (\textsf{down-prim-rec})$$

where n *could* be a free variable in Q. They refer to it as 'going down' since $Q(n+1)$ is needed to deduce $Q(n)$.

The idea is that the property $\forall n P(n)$ is proved using a predicate $Q(m)$ such that $Q(0)$ reduces to $P(n)$. This induction principle is then applied to $Q(0)$. The challenging point here is that a kind of eureka step is required in order to find a satisfactory predicate Q. So, given the proof of $Q(0)$ in terms of $Z^{Q(n)}$ and $S^{\forall m(Q(m+1)\to Q(m))}$, we prove $\forall n P(n)$ by

$$\dfrac{\dfrac{\dfrac{\begin{matrix}|R \\ P(n)\end{matrix}}{Q(0) \to P(n)} \to^+ \qquad \begin{matrix}\vdots \\ Q(0)\end{matrix}}{P(n)} \to^-}{\forall n P(n)} \ \forall^+$$

Here we require the normalization of the code extracted from the proof term $\lambda u^{Q(0)} R^{P(n)}$ to be equal to the identity function. This is because we assume $Q(z)$ to be a predicate that, when instantiated with 0, can be rewritten into $P(n)$ in a finite number of steps, using an opportune set of rewriting rules. This process of simplification is performed using the following, and only the following, axiom:

$$\textsf{Eq-Compat} : \forall x_1, x_2 (x_1 \leadsto x_2 \to P(x_1) \to P(x_2))$$

where $\leadsto$ denotes a binary relation and P a generic predicate symbol. This axiom says that, if we know that a given term (bounded by x_1) is in relation with another term (bounded by x_2) – for example the equality relation – and we know that $P(x_1)$ holds, then we can conclude that $P(x_2)$ holds. Letting the computational content of the Eq-compat axiom be the identity function, it is clear that the program extracted from nested applications of Eq-compact, once normalized, will correspond to the identity function. Since the derivation above is a *detour*, we rewrite it in the following way:

$$\dfrac{\begin{matrix}\vdots \\ Q(0) \\ |R \\ P(n)\end{matrix}}{\forall n P(n)} \ \forall^+$$

which can be read as the replacement of each open assumption $u^{Q(0)}$ in R by the proof of $Q(0)$. The program extracted from the complete proof of $\forall n P(n)$ is the functional `Down.prim_rec` in Figure 4, where `z` *could* depend on `n` (hence the order of the parameters).

```
structure Down
= struct
    (*  prim_rec : int -> 'a * (int -> 'a -> 'a) -> 'a  *)
    fun prim_rec n (z, s)
        = let fun visit m
                    = if m = n then z else s m (visit (m + 1))
          in visit 0
          end

    (*  prim_iter : int -> 'a * ('a -> 'a) -> 'a  *)
    fun prim_iter n (z, s)
        = let fun visit m
                    = if m = n then z else s (visit (m + 1))
          in visit 0
          end
  end
```

Fig. 4. Synthesized functionals for down induction

We now return to the factorial function over natural numbers:

$$\mathsf{fact}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times \mathsf{fact}(n-1) & \text{if } n > 0 \end{cases}$$

Let us prove that $\forall n \exists m(m = \mathsf{fact}(n))$ by going-down primitive recursion. We assume n. In order to prove $\exists m(m = \mathsf{fact}(n))$, we design the new goal $\exists m(\mathsf{fact}(0) \times m = \mathsf{fact}(n))$. Applying the going-down primitive recursive induction principle to this formula requires us to prove the following two subgoals:

—$\exists m(\mathsf{fact}(n) \times m = \mathsf{fact}(n))$: It is sufficient to set $m = 1$.

—Now assume y and $\mathsf{ih} : \exists m(\mathsf{fact}(y+1) \times m = \mathsf{fact}(n))$. We prove $\exists m(\mathsf{fact}(y) \times m = \mathsf{fact}(n))$. By ih, we know that there does exist an m' such that $\mathsf{fact}(y+1) \times m' = \mathsf{fact}(n)$. Considering that $\mathsf{fact}(y+1) = (y+1) \times \mathsf{fact}(y)$, the thesis is proved for $m = (y+1) \times m'$.

The program extracted from this proof reads as follows:

```
fun down_prim_rec_fact n
    = let fun visit m
                = if m = n then 1 else (m + 1) * (visit (m + 1))
      in visit 0
      end
```

This residual program was obtained as an instance of `Down.prim_rec` where `z` is instantiated with the neutral element for multiplication and `s` with the (curried) multiplication function – the same instantiation as in Section 3.1:

```
fun fact n
    = Down.prim_rec n (1, fn i => fn c => (i + 1) * c)
```

3.4 Down primitive iterative induction

Here is the proof principle for primitive iteration:

$$\frac{\begin{matrix} |Z \\ Q(n) \end{matrix} \qquad \begin{matrix} |S \\ \forall^{\mathsf{nc}} m(Q(m+1) \to Q(m)) \end{matrix}}{Q(0)} \;(\text{down-prim-iter})$$

Again, the difference between primitive and iterative induction is that in the iterative case, we quantify non computationally over m in the inductive step. One can then synthesize the functional for down primitive iteration in Figure 4, where `n`, in the local definition of `visit`, is free.

Again, to define the factorial function as an instance of `Down.prim_iter`, we use Kleene's trick. As in Section 3.2, we instantiate `z` with `(1, 1)` and `s` with `fn (i, c) => (i + 1, i * c)` in `Down.prim_iter`:

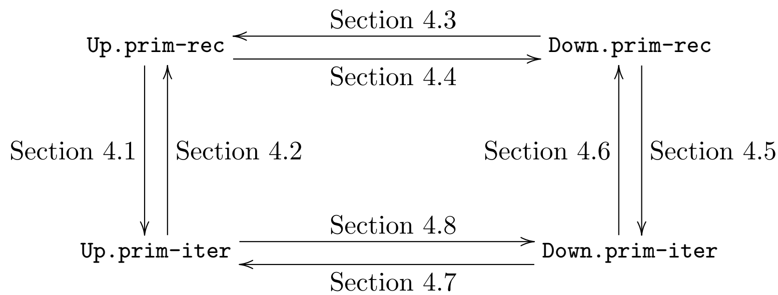
```
fun fact n
    = let val (i, c) = Down.prim_iter n ((1, 1),
                                         fn (i, c) => (i + 1, i * c))
      in c
      end
```

The extracted program reads as follows:

```
fun down_prim_iter_fact n
    = let fun visit m
            = if m = n then (1, 1) else let val (i, c) = visit (m + 1)
                                        in (i + 1, i * c)
                                        end
      in let val (i, c) = visit 0 in c end
      end
```

4. EXPRESSIVE POWER OF THE UP AND DOWN INDUCTION PRINCIPLES

In this section we show that the induction principles (and the associate synthesized functionals) reviewed in Section 3 share the same expressive power.



In the picture above each arrow $A \longrightarrow B$ states the existence of a proof to simulate the induction principle B in terms of the induction principle A. At the level of programs, this corresponds to adapting the functional associated with the principle A in order to simulate the execution of the functional associated with the

principle B. The binary relation $\longrightarrow$ is not transitive, and the missing arrows in the diagram are left as an exercise.

To the knowledge of the authors, the proofs presented in this section and in Section 5 are new. They are available in the first author's home page.[2] By the Curry-Howard correspondence, we manipulated proofs of programs – and not directly their code. This made it possible for us to supply a complete set of tactics to transform the proof of each induction principle into a certified version of the others.

4.1 Up primitive iteration in terms of up primitive recursion

To simulate up primitive iteration in terms of up primitive recursion, we instantiate the base and step of `Up.prim_rec` respectively with `z` and with `fn i => fn c => s c`, where `z` and `s` are the base and step of `up_prim_iter`:

```
fun up_prim_iter (z, s) n
    = let fun visit m
              = if m = 0 then z else s (visit (m - 1))
      in visit n
      end
(*  = Up.prim_rec (z, fn i => fn c => s c) n  *)
```

Proof interpretation:

PROPOSITION 4.1. *Given the proof*

$$\frac{\begin{array}{c}|M\\P(0)\end{array}\qquad\begin{array}{c}|N\\\forall^{\mathsf{nc}}n(P(n)\to P(n+1))\end{array}}{\forall n P(n)}\;\textit{(up-prim-iter)}$$

then there exist M', N' such that:

$$\frac{\begin{array}{c}|M'\\P(0)\end{array}\qquad\begin{array}{c}|N'\\\forall n(P(n)\to P(n+1))\end{array}}{\forall n P(n)}\;\textit{(up-prim-rec)}$$

with computational content equal to `up_prim_iter`.

PROOF.

$$\frac{\begin{array}{c}|M\\P(0)\end{array}\qquad \dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c}|N\\\forall^{\mathsf{nc}}n(P(n)\to P(n+1))\end{array}\qquad n}{P(n)\to P(n+1)}\,\forall^-\qquad [u:P(n)]}{P(n+1)}\,\to^-}{P(n)\to P(n+1)}\,\to^+_u}{\forall n(P(n)\to P(n+1))}\,\forall^+}{\forall n P(n)}\;\text{(up-prim-rec)}$$

□

[2] `http://cmt.math.unipr.it/luca/MinlogCode/`

4.2 Up primitive recursion in terms of up primitive iteration

To simulate up primitive recursion in terms of up primitive iteration, we use Kleene's trick: we instantiate the base and step of `Up.prim_iter` respectively with `(0, z)` and `fn (j, c) => (j + 1, s' j c)`, where `z` and `s` are the base and step of `up_prim_rec`:

```
fun up_prim_rec (z, s) n
    = let fun visit m
              = if m = 0 then (0, z) else let val (i, c) = visit (m - 1)
                                          in (i + 1, s i c)
                                          end
      in let val (i, c) = visit n in c end
      end
(*  = #2 (Up.prim_iter ((0, z), fn (i, c) => (i + 1, s' i c)) n)  *)
```

Proof interpretation:

PROPOSITION 4.2. *Given the proof*

$$\frac{\begin{matrix}|M \\ P(0)\end{matrix} \qquad \begin{matrix}|N \\ \forall n(P(n) \to P(n+1))\end{matrix}}{\forall n P(n)} \text{ (up-prim-rec)}$$

then there exist M', N', R such that:

$$\begin{matrix}\dfrac{\begin{matrix}|M' \\ \exists y(y=0) \wedge P(0)\end{matrix} \qquad \begin{matrix}|N' \\ \forall^{nc} n(\exists y(y=n) \wedge P(n) \to \exists y(y=n+1) \wedge P(n+1))\end{matrix}}{\forall n(\exists y(y=n) \wedge P(n))} \text{ (up-prim-iter)} \\ |R \\ \forall n P(n)\end{matrix}$$

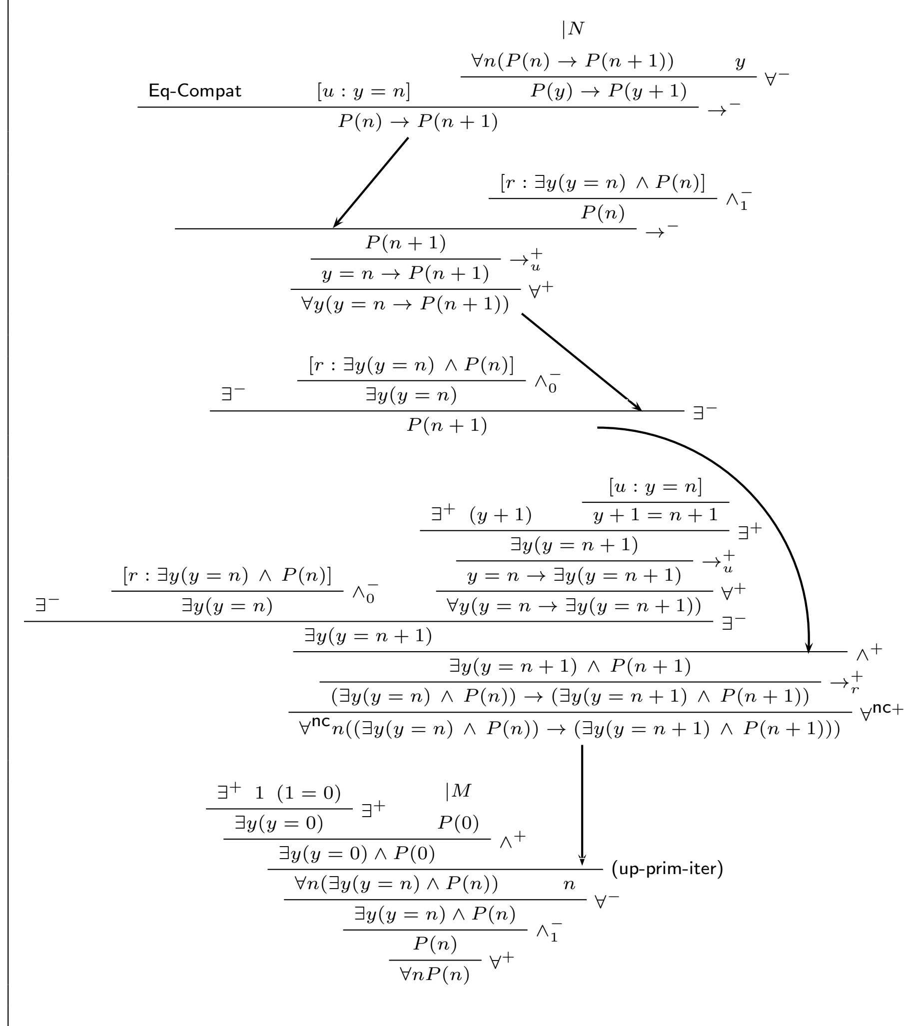
and from which it is possible to extract `up-prim-rec`.

PROOF. See Figure 5. □

4.3 Up primitive recursion in terms of down primitive recursion

To simulate up primitive recursion in terms of down primitive recursion, again we use Kleene's trick: we instantiate the base and step of `Down.prim_rec` respectively with `(0, z)` and `fn m => fn (i, c) => (i + 1, s i c)`, where `z` and `s` are the base and step of `up_prim_rec'`:

```
fun up_prim_rec' (z, s) n
    = let fun visit m
          = if m = n then (0, z) else let val (i, c) = visit (m + 1)
                                      in (i + 1, s i c)
                                      end
      in let val (i, c) = visit 0 in c end
      end
(*  = #2 (Down.prim_rec n ((0, z), fn m => fn (i, c) => (i + 1, s i c)))  *)
```



The variable n does not occur in the content of the proof of the formula $(\exists y(y = n) \wedge P(n)) \to (\exists y(y = n+1) \wedge P(n+1))$, thus the $(\forall^{nc+})$ inference yields a result that is correct w.r.t. the definition given in Figure 2.

Fig. 5. Simulation of up-prim-rec in term of up-prim-iter

Proof interpretation:

PROPOSITION 4.3. *Given the proof*

$$\dfrac{\begin{array}{c}|M\\ P(0)\end{array} \qquad \begin{array}{c}|N\\ \forall n(P(n)\rightarrow P(n+1))\end{array}}{\forall n P(n)}\ \textit{(up-prim-rec)}$$

then there exist M', N' such that:

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}|M'\\ \exists z(z=n-n)\wedge P(n-n)\end{array} \qquad \begin{array}{c}|N'\\ \forall y((\exists z(z=n-(y+1))\wedge P(n-(y+1)))\rightarrow\\ (\exists z(z=n-y)\wedge P(n-y)))\end{array}}{\exists z(z=n-0)\wedge P(n-0)}\ \textit{(down-prim-rec)}}{P(n)}\ \wedge_1^-}{\forall n P(n)}\ \forall^+$$

and from which it is possible to extract the procedure `up-prim-rec'`.

PROOF. See Figure 6. [*Note that the derivation* $^{(*)}$ *in Figure 6 is not true for general* z, n *and* y *in* $\mathbf{N}$. *In order to make* $^{(*)}$ *correct we can replace the end formula on which we apply the* (down-prim-rec) *principle with* $\exists z(z=n-0)\wedge P(n-0)\wedge(n\geq 0)$. *By this replacement, from the assumption* $u: z=n-(y+1)$ *and* $n\geq(y+1)$ *we correctly derive* $z+1=n-y$.] □

4.4 Down primitive recursion in terms of up primitive recursion

To simulate down primitive recursion in terms of up primitive recursion, we instantiate the base and step of `Up.prim_rec` respectively with `(n, z)` (for some input parameter `n`) and `fn m => fn (i, c) => (i - 1, s (i - 1) c)`, where `z` and `s` are the base and step of `down_prim_rec`:

```
fun down_prim_rec (z, s) n
    = let fun visit m
              = if m = 0 then (n, z) else let val (i, c) = visit (m - 1)
                                          in (i - 1, s (i - 1) c)
                                          end
      in let val (i, c) = visit n in c end
      end
(*  = #2 (Up.prim_rec ((n, z),
                       fn m => fn (i, c) => (i - 1, s (i - 1) c)) n)  *)
```

Proof interpretation:

PROPOSITION 4.4. *Given*

$$\dfrac{\dfrac{\begin{array}{c}|M\\ Q(n)\end{array} \qquad \begin{array}{c}|N\\ \forall m(Q(m+1)\rightarrow Q(m))\end{array}}{Q(0)}\ \textit{(down-prim-rec)}}{\begin{array}{c}|R\\ P(n)\end{array}}$$

$$\dfrac{P(n)}{\forall n P(n)}\ \forall^+$$

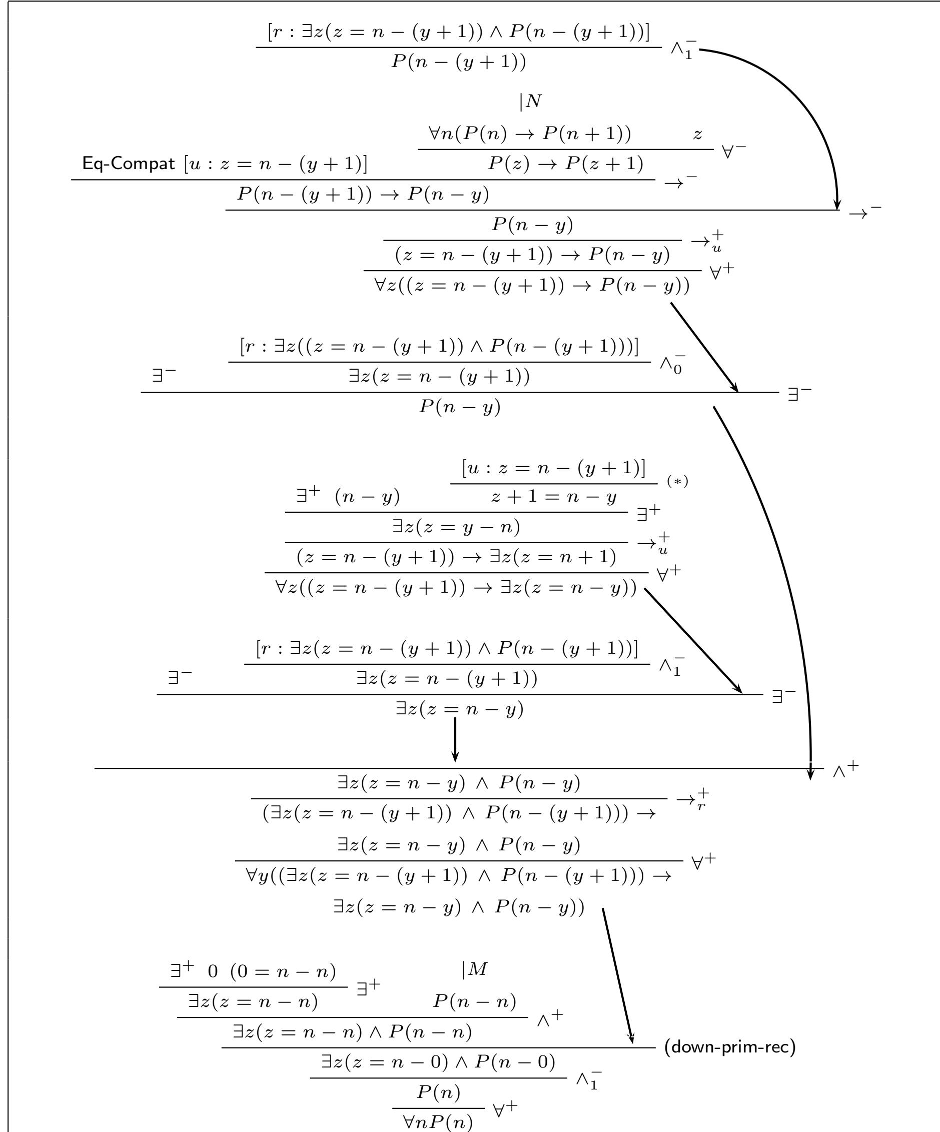


Fig. 6. Simulation of up-prim-rec in term of down-prim-rec

we can find the opportune M', N' such that if the proof of $Q(0)$ is substituted by

$$\dfrac{\dfrac{\dfrac{\begin{matrix}|M' \\ \exists z(z = n) \wedge Q(n)\end{matrix} \quad \begin{matrix}|N' \\ \forall y(((\exists z(z = n - y) \wedge Q(n - y)) \to \\ (\exists z(z = n - (y + 1)) \wedge Q(n - (y + 1))))\end{matrix}}{\forall y(\exists z(z = n - y) \wedge Q(n - y))}\text{(up-prim-rec)} \quad n}{\exists z(z = 0) \wedge Q(0)}\forall^-}{Q(0)}\wedge_1^-$$

then the computational content of the resulting proof corresponds to `down_prim_rec`*.*

Proof. We propose only a sketch because the structure of the proof is the same as the one displayed in Figure 6. The idea is to prove the lemma

$$\forall y(\exists z(z = n - y) \wedge Q(n - y))$$

by up primitive recursion:

Base $y = 0$. We must prove $\exists z(z = n) \wedge Q(n)$. The left conjunct is proved by introducing n for z. The right conjunct is given by M.

Step $y + 1$. Let us assume y and z' such that $z' = n - y$ and $Q(n - y)$. We must prove $\exists z(z = n - (y + 1)) \wedge Q(n - (y + 1))$. The left conjunct is proved introducing $z' - 1$ for z. The right conjunct is proved by instantiating N on $z' - 1$, from which we deduce $Q(z') \to Q(z' - 1)$ that can be rewritten as $Q(n - y) \to Q(n - y - 1)$ by the induction hypothesis $z' = n - y$. We finally instantiate this formula with $Q(n - y)$.

□

4.5 Down primitive iteration in terms of down primitive recursion

To simulate down primitive iteration in terms of down primitive recursion, we instantiate the base and step of `Down.prim_rec` respectively with `z` and `fn i => fn c => s c`, where `z` and `s` are the base and step of `down_prim_iter`:

```
fun down_prim_iter n (z, s)
    = let fun visit m
              = if m = n then z else s (visit (m + 1))
      in visit 0
      end
(*  = Down.prim_rec n (z, fn i => fn c => s c)  *)
```

Proof interpretation:

Proposition 4.5. *Given*

$$\dfrac{\begin{matrix} \dfrac{\begin{matrix}|M \\ Q(n)\end{matrix} \quad \begin{matrix}|N \\ \forall^{nc} m(Q(m + 1) \to Q(m))\end{matrix}}{Q(0)}\text{(down-prim-iter)} \\ |R \\ P(n)\end{matrix}}{\forall n P(n)}\forall^+$$

we can find the opportune M', N' such that if the proof of $Q(0)$ is replaced by

$$\dfrac{\begin{matrix} |M' & & |N' \\ Q(n) & & \forall m(Q(m+1) \to Q(m)) \end{matrix}}{Q(0)}\ \textit{(down-prim-rec)}$$

then the computational content of the transformed proof is equal to `down_prim_iter`*.*

Proof. The structure of the proof is similar to that of Proposition 4.1. We simply set N' to be equal to the proof term $\lambda m, u^{Q(m+1)}(N^{\forall^{\mathsf{nc}} m(Q(m+1)\to Q(m))}\, m\, u)$ and M' to be equal to M. □

4.6 Down primitive recursion in terms of down primitive iteration

To simulate down primitive recursion in terms of down primitive iteration, we instantiate the base and step of `Down.prim_iter` respectively with `(n, z)` (for some given `n`) and `fn (i, c) => (i - 1, s (i - 1) c)`, where `z` and `s` are the base and step of `down_prim_rec'`:

```
fun down_prim_rec' n (z, s)
    = let fun visit m
                = if m = n then (n, z) else let val (i, c) = visit (m + 1)
                                            in (i - 1, s (i - 1) c)
                                            end
      in let val (i, c) = visit 0 in c end
      end
(*  = #2 (Down.prim_iter n ((n, z), fn (i, c) => (i - 1, s (i - 1) c)))  *)
```

Proof interpretation:

Proposition 4.6. *Given the proof*

$$\dfrac{\dfrac{\begin{matrix} |M & & |N \\ Q(n) & & \forall m(Q(m+1) \to Q(m)) \end{matrix}}{\begin{matrix} Q(0) \\ |R \\ P(n) \end{matrix}}\ \textit{(down-prim-rec)}}{\forall n P(n)}\ \forall^+$$

find M', N' and an appropriate Q' such that

$$\dfrac{\dfrac{\begin{matrix} |M' & & |N' \\ Q'(n) & & \forall^{nc} m(Q'(m+1) \to Q'(m)) \end{matrix}}{\begin{matrix} Q'(0) \\ |R \\ P(n) \end{matrix}}\ \textit{(down-prim-iter)}}{\forall n P(n)}\ \forall^+$$

and from which it is possible to extract `down_prim_rec'`*.*

Proof. We propose only a sketch because the structure of the proof is the same as the one displayed in Figure 5. The idea is to set

$$Q'(0) \equiv \exists y (y = 0) \wedge Q(0)$$

and prove $Q'(0)$ by up primitive iteration:

Case n. We have to prove $\exists y (y = n) \wedge Q(n)$, which follows directly by $n = n$ and $M^{Q(n)}$.

Case $m + 1 \rightarrow m$. Assume m (which we quantify non-computationally) and y' such that $y' = m + 1$ and $Q(m + 1)$. We prove $\exists y (y = m) \wedge Q(m)$. For the left conjunct, it is enough to introduce $y' - 1$ for y. For the right conjunct, we need to instantiate N with $(y' - 1)$, obtaining $Q(y') \rightarrow Q(y' - 1)$. By the assumption $y' = m + 1$, we have $Q(m + 1) \rightarrow Q(m)$ and instantiating it with $Q(m + 1)$ we obtain the thesis.

□

4.7 Up primitive iteration in terms of down primitive iteration

To simulate up primitive iteration in terms of down primitive iteration, we instantiate the base and step of `Down.prim_iter` respectively with `(0, z)` and `fn (i, c) => (i + 1, s c)`, where `z` and `s` are the base and step of `up_prim_iter'`:

```
fun up_prim_iter' (z, s) n
      = let fun visit m
            = if m = n then (0, z) else let val (i, c) = visit (m + 1)
                                         in (i + 1, s c)
                                         end
        in let val (i, c) = visit 0 in c end
        end
(*  = #2 (Down.prim_iter n ((0, z), fn (i, c) => (i + 1, s c)))  *)
```

This case is treated as the one in Section 4.4.

4.8 Down primitive iteration in terms up primitive iteration

To simulate down primitive iteration in terms of up primitive iteration, we use Kleene's trick: we instantiate the base and step of `Up.prim_iter` respectively with `(n, z)` and `fn (i, c) => (i - 1, s c)`, where `z` and `s` are the base and step of `down_prim_iter'`:

```
fun down_prim_iter' n (z, s)
    = let fun visit m
              = if m = 0 then (n, z) else let val (i, c) = visit (m - 1)
                                          in (i - 1, s c)
                                          end
      in let val (i, c) = visit n in c end
      end
(*  = #2 (Up.prim_iter ((n, z), fn (i, c) => (i - 1, s c)) n)  *)
```

This case is treated as the one in Section 4.3.

5. PRIMITIVE RECURSION AND ITERATION WITH ACCUMULATORS

Here we present the proof-theoretical analogue of fold-left from functional programming with lists, where the result is accumulated at call time instead of at return time. We consider in turn the accumulator-based versions of each of the induction principles reviewed in Section 3.

5.1 Up primitive recursion with accumulator

Here the problem is how to transform the following up primitive recursive induction principle,

$$\dfrac{\begin{array}{c}|M\\P(0)\end{array} \qquad \begin{array}{c}|N\\\forall n(P(n) \to P(n+1))\end{array}}{\forall n P(n)}\ \text{(up-prim-rec)}$$

into another proof (of the same formula $\forall n P(n)$) but with a computational content that is the accumulator-based version of up primitive recursion:

```
fun up_prim_rec_acc (z, s) n
    = let fun visit m i a
              = if m = 0 then a else visit (m - 1) (i + 1) (s i a)
      in visit n 0 z
      end
```

In these definitions, we use and manipulate two accumulators: i, to count from 0 to n, and a, to store the partial result at step i. Obviously, for $i = n$ we have $a = s\,(n-1)(\ldots(s\,0\,z)\ \ldots)$.

So, given a proof of $\forall n P(n)$, by the up primitive recursive induction principle in terms of $z : M^{P(0)}$ and $s : N^{\forall n(P(n)\to P(n+1))}$, we can build a new proof of $\forall n P(n)$ with content `up_prim_rec_acc` through the following two steps:

(1) We prove the lemma $\forall n \forall m(P(m) \to P(n+m))$ by up primitive recursive induction:
Case $n = 0$. We have to prove

$$\forall m(P(m) \to P(m))$$

which is trivially proved by $(\lambda m, u.u)$.
Case $n + 1$. Let us assume n, the recursive call $p : \forall m(P(m) \to P(n+m))$, m and the accumulator $y : P(m)$. We have to prove

$$P(n + m + 1)$$

We apply s to m and y, obtaining $(s\,m\,y) : P(m + 1)$. Then we apply p to $(m + 1)$ and $s\,m\,y$.

(2) Finally we derive the initial formula $\forall n P(n)$ by assuming n and instantiating the formula proved in the first step on n, 0 and $z : M^{P(0)}$.

5.2 Up primitive iteration with accumulator

We follow the same schema as in Section 5.1. The only difference is that in the intermediate lemma (point 1), we have to quantify non computationally over m. In

other words, we have to prove the modified intermediate lemma:

$$\forall n \forall^{\mathsf{nc}} m (P(m) \to P(n+m))$$

The synthesized program embodies the up primitive iterative induction principle with accumulator:

```
fun up_prim_iter_acc (z, s) n
    = let fun visit m a
              = if m = 0 then a else visit (m - 1) (s a)
      in visit n z
      end
```

5.3 Down primitive recursion with accumulator

Here the problem is how to transform the following down primitive recursive induction principle,

$$\dfrac{\begin{array}{cc} \mid M & \mid N \\ Q(n) & \forall y (Q(y+1) \to Q(y)) \end{array}}{Q(0)} \ (\text{down-prim-rec})$$

into another proof, still of the formula $Q(0)$, but with a computational content that is the accumulator-based version of down primitive recursion:

```
fun down_prim_rec_acc n (z, s)
    = let fun visit m i a
              = if m = n then a else visit (m + 1) (i - 1) (s (i - 1) a)
      in visit 0 n z
      end
```

We propose an approach similar to the one in Section 5.1. We equip the function `down_prim_rec_acc` with two accumulators, denoted by `i` and `a`. The first one is initialized with n at the beginning of the computation and decremented at each iteration. The second one is initialized with z, of type $P(n)$, and is dedicated to store the partial results. The proof from which it is possible to synthesize `up_prim_rec_acc` is based on the following two steps:

(1) We prove the intermediate lemma $\forall i (Q(i) \to Q((i+0)-n))$ by down primitive recursive induction:
Case $y = n$. We have to prove $\forall i (Q(i) \to Q(i))$ that is given, by construction, by the following proof term $\lambda i, u^{Q(i)} u$.
Case $y+1 \to y$. Given y, the induction hypothesis $\mathsf{visit} : \forall i (Q(i) \to Q((i+y+1)-n))$, i and $\mathsf{u} : Q(i)$, we prove $Q((i+y)-n)$ by constructing the following proof term: $(\mathsf{visit}\ (i-1)\ (N^{\forall y (Q(y+1) \to Q(y))}\ (i-1)\ u))^{Q((i+y)-n)}$.

(2) We instantiate the proof of the formula $\forall i (Q(i) \to Q((i+0)-n))$ on n and on $z^{Q(n)}$, obtaining $Q(0)$.

5.4 Down primitive iteration with accumulator

We follow the same schema as in Section 5.3. The only difference is that in the intermediate lemma (point 1), we have to quantify non computationally over i. In other words, we have to prove the modified intermediate lemma:

$$\forall^{\mathsf{nc}} i (Q(i) \to P((i+0)-n))$$

The procedure extracted from this new proof is the following down primitive iteration principle with accumulator:

```
fun down_prim_iter_acc' n (z, s)
    = let fun visit m a
              = if m = n then a else visit (m + 1) (s a)
      in visit 0 z
      end
```

6. CASE STUDY: THE FACTORIAL FUNCTION

In this section we put into practice what we have seen so far, by revisiting the factorial function:

$$\mathsf{fact}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times \mathsf{fact}(n-1) & \text{if } n > 0 \end{cases}$$

We prove $\forall n \exists y (y = \mathsf{fact}\,(n))$ by up primitive induction over natural numbers:

$$\dfrac{\dfrac{\exists^+ \quad 1 = \mathsf{fact}(0)}{\exists y(y = \mathsf{fact}(0))}\,\exists^+ \qquad \dfrac{\dfrac{\dfrac{\exists^- \quad [v : \exists y(y = \mathsf{fact}(n))] \quad \dfrac{\dfrac{\dfrac{\dfrac{[u : y = \mathsf{fact}(n)]}{(n+1) \times y = \mathsf{fact}(n+1)}}{\exists y(y = \mathsf{fact}(n+1))}\,\exists^+}{(y = \mathsf{fact}(n)) \to \exists y(y = \mathsf{fact}(n+1))}\,\to^+_u}{\forall y(y = \mathsf{fact}(n) \to \exists y(y = \mathsf{fact}(n+1)))}\,\forall^+}{\exists y(y = \mathsf{fact}(n+1))}\,\exists^-}{\exists y(y = \mathsf{fact}(n)) \to \exists y(y = \mathsf{fact}(n+1))}\,\to^+_v}{\forall n(\exists y(y = \mathsf{fact}(n)) \to \exists y(y = \mathsf{fact}(n+1)))}\,\forall^+}{\forall n \exists y(y = \mathsf{fact}(n))}\,\text{up-prim-rec}$$

Let us name this proof Proof_fact1. The program extracted from Proof_fact1 reads as follows:

```
fun fact1 n
    = if n = 0 then 1 else n * (fact1 (n - 1))
```

In Proof_fact1 we name B the proof of base and S the proof of step. We have already seen in Section 4.2 how to express at the level of programs, via Kleene's trick, up primitive recursion in terms of up primitive iteration. In the same section, we have seen how to do it also at the level of proofs. So replacing M with B, N with S, and $P(n)$ with $\mathsf{fact}(n)$ in Figure 5, we obtain a new proof, that we name Proof_fact2, with the following computational content:

```
fun fact2 n
    = let fun visit m
              = if m = 0 then (0, 1) else let val (i, c) = visit (m - 1)
                                          in (i + 1, (i + 1) * c)
                                          end
      in let val (i, c) = visit n in c end
      end
```

Proof_fact2 is a proof with the following shape:

$$\dfrac{\dfrac{\dfrac{\dfrac{\begin{matrix}|B\\ \exists y(y=\mathsf{fact}(0))\\ |K\\ \exists y(y=0)\wedge\exists y(y=\mathsf{fact}(0))\end{matrix}\qquad \begin{matrix}|S\\ \forall n(\exists y(y=\mathsf{fact}(n))\to\exists y(y=\mathsf{fact}(n)))\\ |J\\ \forall^{\mathsf{nc}}n((\exists y(y=n)\ \wedge\ \exists y(y=\mathsf{fact}(n)))\to\\ (\exists y(y=n+1)\ \wedge\ \exists y(y=\mathsf{fact}(n+1)))\end{matrix}}{\forall n(\exists y(y=n)\wedge\exists y(y=\mathsf{fact}(n)))}\,(\text{up-prim-iter})\qquad n}{\exists y(y=n)\wedge\exists y(y=\mathsf{fact}(n))}\,\forall^-}{\exists y(y=\mathsf{fact}(n))}\,\wedge_1^-}{\forall n\exists y(y=\mathsf{fact}(n))}\,\forall^+$$

Where $|K$ and $|J$ can be deduced from Figure 5. Now, in Section 5.2, we have seen how to transform an up primitive iterative proof of the form

$$\dfrac{\begin{matrix}|M\\ P(0)\end{matrix}\qquad\begin{matrix}|N\\ \forall^{\mathsf{nc}}n(P(n)\to P(n+1))\end{matrix}}{\forall nP(n)}\,(\text{up-prim-iter})$$

into another proof with an accumulator-based extracted program. Now replacing M with $K[B]$, N with $J[S]$, and $P(n)$ with $\exists y(y=n)\wedge\mathsf{fact}(n)$ in the above schema, and then applying the proof transformation described in Section 5.2 to the proof so instantiated, we obtain a new proof of the formula $\forall n(\exists y(y=n)\wedge\mathsf{fact}(n))$, that we name Proof_fact3. Thus, from the derivation

$$\dfrac{\dfrac{\dfrac{\begin{matrix}\text{Proof_fact3}\\ \forall n(\exists y(y=n)\wedge\exists y(y=\mathsf{fact}(n)))\end{matrix}\qquad n}{\exists y(y=n)\wedge\exists y(y=\mathsf{fact}(n))}\,\forall^-}{\exists y(y=\mathsf{fact}(n))}\,\wedge_1^-}{\forall n\exists y(y=\mathsf{fact}(n))}\,\forall^+$$

we extract the following iterative and accumulator-based version of the factorial function:

```
fun fact3 n
    = let fun visit m (i, c)
              = if m = 0 then (i, c) else visit (m - 1) (i + 1, (i + 1) * c)
      in let val (i, c) = visit n (0, 1) in c end
      end
```

We would like to point out once more that, even if the program obtained after the application of the above transformation is not particularly complicated, our transformation is completely *mechanical* and acts at the level of proofs. The proof itself constitutes a certificate of the correctness of the extracted program.

7. CONCLUSIONS

We have presented a set of mechanical methods to transform the proof of an induction principle that corresponds to a certain recursion principle into a proof of another induction principle that corresponds to another recursion principle. In this work, we considered the well-known up/down primitive and up/down iterative recursion principles and their corresponding proofs. Because each of these transformations was done at the level of proofs, we did not transform programs but proofs of programs: the extracted programs are thus not only correct, but also unadulterated. There is thus no need for adultery to make them run more efficiently. Following the same pattern, we have then transformed each recursive principle into an equivalent one that uses an accumulator. Again, each of the transformations was done at the level of proofs. The extracted programs are tail recursive.

In this article, we have concentrated on the factorial function as our running example, but the up and down induction principles, of course, apply to any program whose number of iterations is bounded by a natural number, as in, e.g., the Euclidean algorithm [3]. To our knowledge, all the proofs here presented are new, and are not treated in the literature. The aim of this work is theoretical, but as illustrated in Section 6, the mechanical methods presented here could find applications in the formal development of programs using proof assistants.

Acknowledgments

This work was carried out in the winter of 2008–2009 while the first author was visiting the second in the Department of Computer Science at Aarhus University [2, Chapter 8]. Thanks are due to the anonymous referees for their comments. The first author is also grateful to Helmut Schwichtenberg for his feedback on a preliminary version of this manuscript, and the second to Mayer Goldberg for introducing him to the generalization of Kleene's trick we have pervasively used here [5].

A. BASICS FOR PROGRAM EXTRACTION FROM CONSTRUCTIVE PROOFS

This appendix contains the material informally presented in Sections 2.3.1, 2.3.2, and 2.3.3.

A.1 Type of a Formula

$$
\begin{aligned}
\tau(P(\vec{x})) &= \begin{cases} \alpha_P & \text{If } P \text{ is a predicate variable with assigned type } \alpha_P \\ \varepsilon & \text{Otherwise} \end{cases} \\
\tau(\exists x^\rho \varphi) &= \begin{cases} \rho & \text{If } \tau(\varphi) = \varepsilon \\ \rho \times \tau(\varphi) & \text{Otherwise} \end{cases} \\
\tau(\forall x^\rho \varphi) &= \begin{cases} \varepsilon & \text{If } \tau(\varphi) = \varepsilon \\ \rho \to \tau(\varphi) & \text{Otherwise} \end{cases} \\
\tau(\exists^{\mathsf{nc}} x^\rho \varphi) &= \tau(\varphi) \\
\tau(\forall^{\mathsf{nc}} x^\rho \varphi) &= \tau(\varphi) \\
\tau(\varphi \wedge \psi) &= \begin{cases} \tau(\varphi) & \text{If } \tau(\psi) = \varepsilon \\ \tau(\psi) & \text{If } \tau(\varphi) = \varepsilon \\ \tau(\varphi) \times \tau(\psi) & \text{Otherwise} \end{cases}
\end{aligned}
$$

$$\tau(\varphi \to \psi) = \begin{cases} \tau(\psi) & \text{If}\, \tau(\varphi) = \varepsilon \\ \varepsilon & \text{If}\, \tau(\psi) = \varepsilon \\ \tau(\varphi) \to \tau(\psi) & \text{Otherwise} \end{cases}$$

A.2 Extraction map

$$\begin{aligned}
[\![u^\varphi]\!] &= x_u^{\tau(\varphi)} \quad (x_u^{\tau(\varphi)} \text{ uniquely associated with } \varphi) \\
[\![\lambda u^\varphi M]\!] &= \begin{cases} [\![M]\!] & \text{If}\, \tau(\varphi) = \varepsilon \\ \lambda x_u^{\tau(\varphi)} [\![M]\!] & \text{Otherwise} \end{cases} \\
[\![M^{\varphi \to \psi} N^\varphi]\!] &= \begin{cases} [\![M]\!] & \text{If}\, \tau(\varphi) = \varepsilon \\ [\![M]\!][\![N]\!] & \text{Otherwise} \end{cases} \\
[\![\langle M^\varphi, N^\psi \rangle]\!] &= \begin{cases} [\![N]\!] & \text{If}\, \tau(\varphi) = \varepsilon \\ [\![M]\!] & \text{If}\, \tau(\psi) = \varepsilon \\ \langle [\![M]\!], [\![N]\!] \rangle & \text{Otherwise} \end{cases} \\
[\![M^{\varphi \wedge \psi} i]\!] &= \begin{cases} [\![M]\!] & \text{If}\, \tau(\varphi) = \varepsilon \,\text{or}\, \tau(\psi) = \varepsilon \\ \pi_i [\![M]\!] & \text{Otherwise} \end{cases} \\
[\![(\lambda x^\rho M)^{\forall x \varphi}]\!] &= \lambda x^\rho [\![M]\!] \\
[\![M^{\forall x \varphi} t]\!] &= [\![M]\!] t \\
[\![(\lambda x^\rho M)^{\forall^{\mathsf{nc}} x \varphi}]\!] &= [\![M]\!] \\
[\![M^{\forall^{\mathsf{nc}} x \varphi} t]\!] &= [\![M]\!]
\end{aligned}$$

Content of the proof constants:

$$\begin{aligned}
[\![\exists^-_{x^\rho, \varphi, \psi}]\!] &= \begin{cases} \lambda x^\rho f^{\rho \to \tau(\psi)}.fx & \text{If } \tau(\varphi) = \varepsilon \\ \lambda x^{\rho \times \tau(\varphi)} f^{\rho \to \tau(\varphi) \to \tau(\psi)}.f(\pi_0 x)(\pi_1 x) & \text{Otherwise} \end{cases} \\
[\![\exists^+_{x^\rho, \varphi}]\!] &= \begin{cases} \lambda x^\rho x & \text{If } \tau(\varphi) = \varepsilon \\ \lambda x^\rho y^{\tau(\varphi)}.\langle x, y \rangle & \text{Otherwise} \end{cases} \\
[\![(\exists^{\mathsf{nc}})^-_{x^\rho, \varphi, \psi}]\!] &= \begin{cases} \lambda x^{\tau(\psi)}.x & \text{If } \tau(\varphi) = \varepsilon \\ \lambda x^{\tau(\varphi)} f^{\tau(\varphi) \to \tau(\psi)}.fx & \text{Otherwise} \end{cases} \\
[\![(\exists^{\mathsf{nc}})^+_{x^\rho, \varphi}]\!] &= \lambda x^{\tau(\varphi)} x \\
[\![\mathsf{IF}_\varphi]\!] &= \lambda b^{\mathbf{B}}, l^{\tau(\varphi)}, r^{\tau(\varphi)}.(\mathsf{if}\, b\, l\, r) \quad \text{If } \tau(\varphi) \neq \varepsilon \\
[\![\mathrm{Ind}_{n, \varphi(n)}]\!] &= \mathcal{R}^\sigma_{\mathbf{N}} \\
[\![\mathrm{Ind}_{l, \varphi(l)}]\!] &= \mathcal{R}^\sigma_{\mathbf{L}(\rho)} \\
[\![\mathrm{Ind}_{t, \varphi(t)}]\!] &= \mathcal{R}^\sigma_{\mathbf{B}}
\end{aligned}$$

A.3 Modified Realizability

$$\begin{aligned}
r\, \mathbf{mr}\, P(\vec{t}) &= P(\vec{t}) \\
r\, \mathbf{mr}\, (\exists x.\varphi) &= \begin{cases} \varepsilon\, \mathbf{mr} \varphi[x/r] & \text{If}\, \tau(\varphi) = \varepsilon \\ \pi_1 r\, \mathbf{mr}\, \varphi[x/\pi_0 r] & \text{Otherwise} \end{cases} \\
r\, \mathbf{mr}\, (\forall x.\varphi) &= \begin{cases} \forall x.\varepsilon\, \mathbf{mr}\, \varphi & \text{If}\, \tau(\varphi) = \varepsilon \\ \forall x.r\, \mathbf{mr}\, \varphi & \text{Otherwise} \end{cases}
\end{aligned}$$

$$r\,\mathbf{mr}\,(\exists^{\mathsf{nc}}x.\varphi) \;=\; \begin{cases} \exists^{\mathsf{nc}}x.\varepsilon\,\mathbf{mr}\varphi & \text{If}\,\tau(\varphi)=\varepsilon \\ \exists^{\mathsf{nc}}x.r\,\mathbf{mr}\,\varphi & \text{Otherwise} \end{cases}$$

$$r\,\mathbf{mr}\,(\forall x^{\mathsf{nc}}.\varphi) \;=\; \begin{cases} \forall^{\mathsf{nc}}x.\varepsilon\,\mathbf{mr}\,\varphi & \text{If}\,\tau(\varphi)=\varepsilon \\ \forall^{\mathsf{nc}}x.r\,\mathbf{mr}\,\varphi & \text{Otherwise} \end{cases}$$

$$r\,\mathbf{mr}\,(\varphi\to\psi) \;=\; \begin{cases} \varepsilon\,\mathbf{mr}\,\varphi\to r\,\mathbf{mr}\,\psi & \text{if}\,\tau(\varphi)=\varepsilon \\ \forall x.x\,\mathbf{mr}\,\varphi\to\varepsilon\,\mathbf{mr}\,\psi & \text{If}\,\tau(\varphi)\neq\varepsilon=\tau(\psi) \\ \forall x.x\,\mathbf{mr}\,\varphi\to rx\,\mathbf{mr}\,\psi & \text{Otherwise} \end{cases}$$

$$r\,\mathbf{mr}\,(\varphi\wedge\psi) \;=\; \begin{cases} \varepsilon\,\mathbf{mr}\,\varphi\wedge r\,\mathbf{mr}\,\psi & \text{If}\,\tau(\varphi)=\varepsilon \\ r\,\mathbf{mr}\,\varphi\to\varepsilon\,\mathbf{mr}\,\psi & \text{If}\,\tau(\psi)=\varepsilon \\ \pi_0 r\,\mathbf{mr}\,\varphi\to\pi_1 r\,\mathbf{mr}\,\psi & \text{Otherwise} \end{cases}$$

B. ON THE ORDER OF MULTIPLICATIONS IN EXTRACTED PROGRAMS

The computation associated to the factorial function can be syntactically characterized with the following data type:

```
datatype residual = LIT of int
                  | TIMES of int * residual
```

For example, the computation associated to the factorial function of Section 3.1 can be visualized by making it map an integer to a syntactic witness of the trail of multiplications it entails:

```
(*  fact_gen : int -> residual  *)
fun fact_gen n
    = Up.prim_rec (LIT 1, fn i => fn c => TIMES (i + 1, c)) n
```

Applying `fact_gen` to `5`, for example, yields the following residual trail:

```
TIMES (5, TIMES (4, TIMES (3, TIMES (2, TIMES (1, LIT 1)))))
```

The same residual trail is obtained for the factorial functions defined in Section 3.2 and in Section 6.

Likewise, the computation associated to the factorial function of Section 3.3 can also be visualized by making it map an integer to a syntactic witness of the trail of multiplications it entails:

```
fun fact_gen n
    = down_prim_rec (LIT 1, fn i => fn c => TIMES (i + 1, c)) n
```

Applying `fact_gen` to `5`, for example, yields the following residual trail:

```
TIMES (1, TIMES (2, TIMES (3, TIMES (4, TIMES (5, LIT 1)))))
```

The same residual trail is obtained for the factorial function defined in Section 3.4.

As can be readily seen, the order of the multiplications is not the same. It does not matter for the factorial function since multiplication is associative and commutative, but it would for other functions, e.g., to process lists.

References

[1] Andrea Asperti and Enrico Tassi. Modified realizability and inductive types. Technical report, Dipartimento di Scienze dell'Informazione, Università degli Studi di Bologna, 2006.

[2] Luca Chiarabini. *Program Development by Proof Transformation*. PhD thesis, Fakultät für Mathematik, Informatik und Statistik, Ludwig Maximilians Universität, München, Germany, 2009.

[3] Olivier Danvy and Mayer Goldberg. Partial evaluation of the Euclidean algorithm. *Lisp and Symbolic Computation*, 10(2):101–111, 1997.

[4] Kurt Gödel. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunkts. *Dialectica*, 12:280–287, 1958.

[5] Mayer Goldberg. *Recursive Application Survival in the λ-Calculus*. PhD thesis, Computer Science Department, Indiana University, Bloomington, Indiana, May 1996.

[6] Stephen C. Kleene. Origins of recursive function theory. *Annals of the History of Computing*, 3(1):52–67, January 1981.

[7] Georg Kreisel. Interpretation of Analysis by means of Functionals of Finite Type. In Arend Heyting, editor, *Constructivity in Mathematics*, 1959.

[8] Zohar Manna and Richard J. Waldinger. Towards automatic program synthesis. *Communications of the ACM*, 14(3), 1971.

[9] Iman Poernomo, John N. Crossley, and Martin Wirsing. *Adapting Proofs-as-Programs : The Curry-Howard Protocol*. Springer, 2005.

[10] Morten Heine B. Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard Isomorphism*. Elsevier, 1998.

[11] Helmut Schwichtenberg. Minimal logic for computable functionals. `http://www.mathematik.uni-muenchen.de/~chiarabi/mlcf.pdf`, February 2008.

[12] Anne S. Troelstra. Constructivism and proof theory, 2003.

[13] Dirk van Dalen. *Logic and Structure*. Springer, 2008.