5. Program Extraction From Constructive Proofs

5.1. **The Type of a Formula.** We assign to every formula A an object $\tau(A)$ (a type or the symbol ε). $\tau(A)$ is intended to be the type of the program to be extracted from a proof of A. In case $\tau(A) = \varepsilon$ proofs of A have no computational content; such formulas A are called *Harrop formulas*.

Recall that we allow free predicate variables, to be viewed as placeholders for formulas (or more precisely, comprehension terms). Since we do not know in advance which formula will be substituted for a predicate variable, we use a type variable as the type of the program to be extracted from a proof of an atom involving a predicate variable. Therefore our definition of $\tau(A)$ is relative to a given assignment of type variables to some (see below) predicate variables.

$$\tau(P(\vec{s})) := \begin{cases} \alpha_P & \text{if } P \text{ is a predicate variable with assigned } \alpha_P \\ \varepsilon & \text{otherwise} \end{cases}$$

$$\tau(\exists x^\rho A) := \begin{cases} \rho & \text{if } \tau(A) = \varepsilon \\ \rho \times \tau(A) & \text{otherwise} \end{cases}$$

$$\tau(\forall x^\rho A) := \begin{cases} \varepsilon & \text{if } \tau(A) = \varepsilon \\ \rho \to \tau(A) & \text{otherwise} \end{cases}$$

$$\tau(A_0 \wedge A_1) := \begin{cases} \tau(A_i) & \text{if } \tau(A_{1-i}) = \varepsilon \\ \tau(A_0) \times \tau(A_1) & \text{otherwise} \end{cases}$$

$$\tau(A \to B) := \begin{cases} \tau(B) & \text{if } \tau(A) = \varepsilon \\ \varepsilon & \text{if } \tau(B) = \varepsilon \\ \tau(A) \to \tau(B) & \text{otherwise} \end{cases}$$

5.2. **The Program Extracted from a Derivation.** We now define, for a given derivation M of a formula A with $\tau(A) \neq \varepsilon$, its *extracted program* $[\![M]\!]$ of type $\tau(A)$.

$$[\![u^A]\!] := x_u^{\tau(A)} \quad (x_u^{\tau(A)} \text{ uniquely associated with } u^A)$$

$$[\![\lambda u^A M]\!] := \begin{cases} [\![M]\!] & \text{if } \tau(A) = \varepsilon \\ \lambda x_u^{\tau(A)} [\![M]\!] & \text{otherwise} \end{cases}$$

$$[\![M^{A \to B} N]\!] := \begin{cases} [\![M]\!] & \text{if } \tau(A) = \varepsilon \\ [\![M]\!][\![N]\!] & \text{otherwise} \end{cases}$$

$$[\![\langle M_0^{A_0}, M_1^{A_1} \rangle]\!] := \begin{cases} [\![M_i]\!] & \text{if } \tau(A_{1-i}) = \varepsilon \\ \langle [\![M_0]\!], [\![M_1]\!] \rangle & \text{otherwise} \end{cases}$$

$$[\![M^{A_0 \wedge A_1} i]\!] := \begin{cases} [\![M]\!] & \text{if } \tau(A_{1-i}) = \varepsilon \\ [\![M]\!] i & \text{otherwise} \end{cases}$$

$$[\![(\lambda x^\rho M)^{\forall x A}]\!] := \lambda x^\rho [\![M]\!]$$

$$[\![M^{\forall x A} t]\!] := [\![M]\!] t$$

We also need extracted programs for induction, cases and $\exists$-axioms; these will be defined below. For derivations M^A where $\tau(A) = \varepsilon$ (i.e. A is a Harrop formula) we define $[\![M]\!] := \varepsilon$ (ε some new symbol). This applies in particular if A is $\exists$-free and contains no predicate variables.

5.2.1. *Extracted Program of an Induction Axiom.* Recall the general form of induction over simultaneous free algebras $\vec{\mu} = \mu\vec{\alpha}\,\vec{\kappa}$, with goal formulas $\forall x_j^{\mu_j} A_j$, from Section 3.2. For the constructor type

$$\kappa_i = \vec{\rho} \to (\vec{\sigma}_1 \to \alpha_{j_1}) \to \cdots \to (\vec{\sigma}_n \to \alpha_{j_n}) \to \alpha_j \in \mathsf{KT}(\vec{\alpha})$$

we have the *step formula*

$$\begin{aligned} D_i := \forall y_1^{\rho_1}, \dots, y_m^{\rho_m}, y_{m+1}^{\vec{\sigma}_1 \to \mu_{j_1}}, \dots, y_{m+n}^{\vec{\sigma}_n \to \mu_{j_n}}.& \\ \forall \vec{x}^{\vec{\sigma}_1} A_{j_1}[x_{j_1} := y_{m+1}\vec{x}] \to \cdots \to& \\ \forall \vec{x}^{\vec{\sigma}_n} A_{j_n}[x_{j_n} := y_{m+n}\vec{x}] \to& \\ A_j[x_j := \mathsf{constr}_i^{\vec{\mu}}(\vec{y})].& \end{aligned} \tag{7}$$

Here $\vec{y} = y_1^{\rho_1}, \dots, y_m^{\rho_m}, y_{m+1}^{\vec{\sigma}_1 \to \mu_{j_1}}, \dots, y_{m+n}^{\vec{\sigma}_n \to \mu_{j_n}}$ are the *components* of the object $\mathsf{constr}_i^{\vec{\mu}}(\vec{y})$ of type μ_j under consideration, and

$$\forall \vec{x}^{\vec{\sigma}_1} A_{j_1}[x_{j_1} := y_{m+1}\vec{x}], \dots, \forall \vec{x}^{\vec{\sigma}_n} A_{j_n}[x_{j_n} := y_{m+n}\vec{x}]$$

are the hypotheses available when proving the induction step. The induction axiom $\mathsf{Ind}_{\mu_j}^{\vec{x},\vec{A}}$ or shortly Ind_j then proves the universal closure of the formula

$$D_1 \to \cdots \to D_k \to \forall x_j^{\mu_j} A_j.$$

$[\![\mathsf{Ind}_j]\!]$ is defined to be the recursion operator $\mathcal{R}_{\mu_j}^{\vec{\mu},\vec{\tau}}$. Here $\vec{\mu}, \vec{\tau}$ list only the types μ_j, τ_j with $\tau_j := \tau(A_j) \neq \varepsilon$, i.e. the recursion operator is simplified accordingly.

Remark. It is possible to use variants of the induction scheme, were some or all of the universal quantifiers in the step formula (7) have no computational content.

Example. For the induction scheme

$$\mathsf{Ind}_{n,A}\colon A[n := 0] \to (\forall n.A \to A[n := n+1]) \to \forall n A$$

we have

$$[\![\mathsf{Ind}_{n,A}]\!] := \mathcal{R}_{\mathbf{N}}^{\tau}\colon \tau \to (\mathbf{N} \to \tau \to \tau) \to \mathbf{N} \to \tau,$$

where $\tau := \tau(A) \neq \varepsilon$.

5.2.2. *Extracted Program of a Cases Axiom.* Recall the cases axioms from Section 3.4. They again refer to simultaneous free algebras $\vec{\mu} = \mu\vec{\alpha}\,\vec{\kappa}$, and goal formulas $\forall x_j^{\mu_j} A_j$. For the constructor type

$$\kappa_i = \vec{\rho} \to (\vec{\sigma}_1 \to \alpha_{j_1}) \to \cdots \to (\vec{\sigma}_n \to \alpha_{j_n}) \to \alpha_j \in \mathsf{KT}(\vec{\alpha})$$

we have the *step formula*

$$D_i := \forall y_1^{\rho_1}, \dots, y_m^{\rho_m}, y_{m+1}^{\vec{\sigma}_1 \to \mu_{j_1}}, \dots, y_{m+n}^{\vec{\sigma}_n \to \mu_{j_n}}.A_j[x_j := \mathsf{constr}_i^{\vec{\mu}}(\vec{y})].$$

Here $\vec{y} = y_1^{\rho_1}, \dots, y_m^{\rho_m}, y_{m+1}^{\vec{\sigma}_1 \to \mu_{j_1}}, \dots, y_{m+n}^{\vec{\sigma}_n \to \mu_{j_n}}$ are the *components* of the object $\mathsf{constr}_i^{\vec{\mu}}(\vec{y})$ of type μ_j under consideration; notice that no induction hypotheses are available. The cases axiom Cases_{x_j,A_j} or shortly Cases_j then proves the universal closure of the formula

$$D_{i_1},\dots,D_{i_q} \to \forall x_j^{\mu_j}\, A_j.$$

where $D_{i_1},\dots,D_{i_q}$ consists of all D_i concerning constructors for μ_j.

$[\![\mathsf{Cases}_j]\!]$ is defined by means of the `if`-construct to be

$$[\![\mathsf{Cases}_j]\!] := \lambda f_1 \dots \lambda f_q \lambda x[\mathtt{if}\ x\ f_1\ \dots\ f_q],$$

provided $\tau(A_j) \neq \varepsilon$.

Example. For the cases axioms

$$\mathsf{Cases}_{n,A}\colon A[n := 0] \to \forall n A[n := \mathsf{S}n] \to \ \forall n^{\mathbf{N}} A,$$
$$\mathsf{Cases}_{l,A}\colon A[l := \mathsf{Nil}] \to \forall x, l A[l := \mathsf{Cons}(x,l)] \to \forall l^{\mathbf{L}(\alpha)} A.$$

we have

$$[\![\mathsf{Cases}_{n,A}]\!] := \lambda f_1 \lambda f_2 \lambda n[\mathtt{if}\ n\ f_1\ f_2],$$
$$[\![\mathsf{Cases}_{l,A}]\!] := \lambda f_1 \lambda f_2 \lambda l[\mathtt{if}\ l\ f_1\ f],$$

where we assume $\tau(A) \neq \varepsilon$.

5.2.3. *Extracted Programs of Existence Axioms.* For the axioms

$$\exists^+_{x,B}\colon \forall x^\rho.A \to \exists x^\rho A \qquad \texttt{Ex-Intro}$$
$$\exists^-_{x,A,B}\colon \exists x^\rho A \to (\forall x^\rho.A \to B) \to B \quad \texttt{Ex-Elim}$$

we set

$$[\![\exists^+_{x^\rho,A}]\!] := \begin{cases} \lambda x^\rho x & \text{if } \tau(A) = \varepsilon \\ \lambda x^\rho \lambda y^{\tau(A)} \langle x, y\rangle & \text{otherwise} \end{cases}$$

$$[\![\exists^-_{x^\rho,A,B}]\!] := \begin{cases} \lambda x^\rho \lambda f^{\rho\to\tau(B)}.fx & \text{if } \tau(A) = \varepsilon \\ \lambda z^{\rho\times\tau(A)} \lambda f^{\rho\to\tau(A)\to\tau(B)}.f(z0)(z1) & \text{otherwise} \end{cases}$$

5.3. **Realizability.** Finally we define the notion of *(modified) realizability*. The term "modified" is used sometimes for historical reasons, to distinguish this form of realizability from the (earlier) Kleene-style realizability. More precisely, we define formulas $r\,\mathbf{r}\,A$, where A is a formula and r is either a term of type $\tau(A)$ if the latter is a type, or the symbol ε if $\tau(A) = \varepsilon$.

5.3.1. *Definition of Realizability.* In order to define realizability for atoms built with predicate variables, we have to provide for every predicate variable P of arity $\vec{\rho}$ with assigned α_P a new predicate variable $P^{\mathbf{r}}$ of arity $\alpha_P, \vec{\rho}$.

$$r\,\mathbf{r}\,P(\vec{s}) \;=\; \begin{cases} P^{\mathbf{r}}(r,\vec{s}) & \text{if } P \text{ is a predicate variable with assigned } \alpha_P \\ P(\vec{s}) & \text{if } P \text{ is a predicate constant} \end{cases}$$

$$r\,\mathbf{r}\,(\exists x A) \;=\; \begin{cases} \varepsilon\,\mathbf{r}\,A[x := r] & \text{if } \tau(A) = \varepsilon \\ r1\,\mathbf{r}\,A[x := r0] & \text{otherwise} \end{cases}$$

$$r\,\mathbf{r}\,(\forall x A) \;=\; \begin{cases} \forall x.\varepsilon\,\mathbf{r}\,A & \text{if } \tau(A) = \varepsilon \\ \forall x.rx\,\mathbf{r}\,A & \text{otherwise} \end{cases}$$

$$r\,\mathbf{r}\,(A \to B) \;=\; \begin{cases} \varepsilon\,\mathbf{r}\,A \;\to\; r\,\mathbf{r}\,B & \text{if } \tau(A)=\varepsilon \\ \forall x.x\,\mathbf{r}\,A \;\to\; \varepsilon\,\mathbf{r}\,B & \text{if } \tau(A)\neq\varepsilon=\tau(B) \\ \forall x.x\,\mathbf{r}\,A \;\to\; rx\,\mathbf{r}\,B & \text{otherwise} \end{cases}$$

$$r\,\mathbf{r}\,(A_0 \wedge A_1) \;=\; \begin{cases} \varepsilon\,\mathbf{r}\,A_0 \;\wedge\; r\,\mathbf{r}\,A_1 & \text{if } \tau(A_0)=\varepsilon \\ r\,\mathbf{r}\,A_0 \;\wedge\; \varepsilon\,\mathbf{r}\,A_1 & \text{if } \tau(A_1)=\varepsilon \\ r0\,\mathbf{r}\,A_0 \;\wedge\; r1\,\mathbf{r}\,A_1 & \text{otherwise} \end{cases}$$

Formulas which do not contain the existence quantifier $\exists$ play a special role in this context; we call them $\exists$-*free* (or *invariant*) formulas; in the literature such formulas are also called “negative”. Their crucial property is that for an $\exists$-free formula A without predicate variables P with assigned α_P we have $\tau(A)=\varepsilon$ and $\varepsilon\,\mathbf{r}\,A = A$. In particular, we have to assign an α_P to every predicate variable P of “existence degree” $\neq 0$, which means that it can be substituted by a formula containing $\exists$ or other predicate variables Q with assigned α_Q.

For the formulation of the soundness theorem it is useful to let $x_u := \varepsilon$ if u^A is an assumption variable with a Harrop formula A.

Theorem (Soundness)**.** *If M is a derivation of a formula B, then there is a derivation $\mu(M)$ of $[\![M]\!]\,\mathbf{r}\,B$ from $\{\, x_u\,\mathbf{r}\,C \mid u^C \in \mathsf{FA}(M)\,\}$.*

Proof. Induction on M. □

5.4. **A Case Studies: Fibonacci Numbers.** Let α_n be the n-th Fibonacci number, i.e.

$$\begin{aligned} \alpha_0 &:= 0 \\ \alpha_1 &:= 1 \\ \alpha_n &:= \alpha_{n-2} + \alpha_{n-1} \quad \text{for } n \geq 2 \end{aligned}$$

Here is a naive SCHEME-program to compute them

```
(define (f n)
  (if (= n 0)
      0
      (if (= 1 n)
          1
          (+ (f (- n 2)) (f (- n 1))))))
```

Clearly this program is very inefficient, since it leads to many recomputations. Our goal here is to demonstrate that deriving a program from a constructive proof of the existence of Fibonacci numbers avoids the immediate source of the inefficiency.

We first build a constructive existence proof for the Fibonacci numbers.

```
(set-goal
  (pf "G 0 0 -> G 1 1 ->
       (all n,k,l.G n k -> G(n+1)l -> G(n+2)(k+l)) ->
       all n ex k,l. G n k & G(n+1)l"))
(assume "Init-Zero" "Init-One" "Step")
(ind)
```

```
; Base
(ex-intro (pt "0"))
(ex-intro (pt "1"))
(prop)

; Step
(assume "n" "IH")
(by-assume-with "IH" "k" "IH-k")
(by-assume-with "IH-k" "l" "IH-l")
(ex-intro (pt "l"))
(ex-intro (pt "k+l"))
(search)
```

The extracted program can now be obtained as follows.

```
(define Fib-neterm
  (nt (proof-to-extracted-term
    (theorem-name-to-proof "Fib"))))
```

The extracted term is obtained by

```
(term-to-string Fib-neterm)
```

which yields

```
(Rec nat=>nat@@nat)(0@1)([n1,p2]right p2@left p2+right p2)
```

This clearly is a linear algorithm. To run it, type

```
(pp (nt (make-term-in-app-form Fib-neterm (pt "13"))))
```

which yields

```
233@377
```

We can also use an "external" extraction, yielding Scheme code:

```
(term-to-expr Fib-neterm)
```

produces

```
((natrec (cons 0 1))
 (lambda (n1)
   (lambda (p2) (cons (cdr p2) (+ (car p2) (cdr p2))))))
```

To run this code, we need to give a Scheme-definition of `natrec`, that is recursion on the natural numbers:

```
(define (natrec init)
  (lambda (step)
    (lambda (n)
      (if (= 0 n)
          init
          ((step n) (((natrec init) step) (- n 1)))))))
```

Remark. There are other algorithms to compute the Fibonacci numbers α_n, which run in logarithmic time. By definition we have

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} \alpha_{n-2} \\ \alpha_{n-1} \end{pmatrix} = \begin{pmatrix} \alpha_{n-1} \\ \alpha_{n-2} + \alpha_{n-1} \end{pmatrix} = \begin{pmatrix} \alpha_{n-1} \\ \alpha_n \end{pmatrix},$$

hence with $A = \left(\begin{smallmatrix} 0 & 1 \\ 1 & 1 \end{smallmatrix}\right) \in \mathbb{R}^{2\times 2}$

$$A^n \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} \alpha_n \\ \alpha_{n+1} \end{pmatrix}.$$

So an easy way to compute the Fibonacci numbers is by computing the powers of A. This can be done in time $O(\log(n))$, since

$$A^{2n} = (A^n)^2$$
$$A^{2n+1} = A^{2n} \cdot A$$

It is possible to obtain this algorithm as computational content of a proof: Use u, v to denote vectors in $\mathbb{Z}^2$ and X, Y to denote matrices in $\mathbb{Z}^{2\times 2}$. Let $G(n, u)$ mean that u is the vector of the n-th and $(n + 1)$-th Fibonacci number. G can be axiomatized by

$$G(1, \begin{pmatrix} 1 \\ 1 \end{pmatrix}), \quad G(n, \begin{pmatrix} \alpha \\ \beta \end{pmatrix}) \to G(n + 1, \begin{pmatrix} \beta \\ \alpha + \beta \end{pmatrix}).$$

Then prove by induction on the positive numbers n

$$\forall n \exists X \forall m, u. G(m, u) \to G(m + n, Xu).$$

Clearly this X must be A^n.