Program extraction from normalization proofs

Ulrich Berger, Stefan Berghofer,
Pierre Letouzey and Helmut Schwichtenberg

Abstract

This paper describes formalizations of Tait's normalization proof for the simply typed λ-calculus in the proof assistants Minlog, Coq and Isabelle/HOL. From the formal proofs programs are machine-extracted that implement variants of the well-known normalization-by-evaluation algorithm. The case study is used to test and compare the program extraction machineries of the three proof assistants in a non-trivial setting.

1 Introduction

We formalize a version of Tait's normalization proof for the simply typed λ-calculus and machine-extract a program which turns out to implement the well-known normalization-by-evaluation algorithm [6]. On paper, this has already been done by the first author in his contribution [4] to TLCA 1993. However – as is to be expected – the formalization turned out to be not at all a trivial matter, so that it appears to be worthwhile to describe some of the choices which have simplified the task considerably. On the other hand, a full formalization of the proof *is* necessary for machine extraction of a program.

Here we carry this out for the proof assistants *Minlog*, *Coq* and *Isabelle/HOL* which all have suitable program extraction machineries built in. This provides a useful occasion to test these machineries in a non-trivial setting and to compare the three proof assistants. The crucial questions, on which the formalizations diverge, are how to (1) model the simply typed λ-calculus, (2) represent in the given logical system the notions introduced in the proof (i.p. Tait's strong computability predicates), (3) optimize program extraction in order to get as close as possible to normalization-by-evaluation.

The paper is organized as follows. In Section 2 we recall Tait's approach – via so-called strong computability predicates – to a normalization proof and sketch the normalization-by-evaluation algorithm contained in that proof,

following [4]. Section 3 describes two *Minlog formalizations*. The first proves weak normalization for λ-terms in de Bruijn notation with a Church-style typing, the second strong normalization for λ-terms with named variables and a Curry-style typing. In order to obtain optimized programs the formalizations use two versions of quantifiers, with and without computational content. Another important detail is the fact that Tait's notion of computability of a term r, $\mathrm{SC}_\rho(r)$, is defined as $\exists a \mathrm{SCr}_\rho(a, r)$ where the relation $\mathrm{SCr}_\rho(a, r)$ is axiomatized such that it expresses "a realizes $\mathrm{SC}_\rho(r)$". This has the effect that the computational content of Lemma 2 (stating that computability is closed under reverse head reduction, see Section 2) is the identity. The *Coq formalization* in Section 4 uses, like the first Minlog formalization, de Bruijn notation and Church-style typing, but takes advantage of the fact that Coq implements dependent type theory and thus is able to *define* Tait's computability predicate (in contrast, both Minlog formalization need to describe these predicates axiomatically). This formalization is complete in the sense that all auxiliary lemmas are proven (unlike the Minlog and Isabelle/HOL formalizations which take some computationally irrelevant lemmas as axioms). In fact, two formalizations are given, yielding programs for the long respectively short η-normal form. The *Isabelle/HOL formalization* is described in Section 5; it is based on the second Minlog approach. The complete proof scripts are available at `www.minlog-system.de` in the directory `examples/tait`, `www.lri.fr/~letouzey/download/tait.tgz` and `www4.in.tum.de/~berghofe/papers/Tait/`.

Related normalization algorithms have been machine-extracted from formal proofs in the type-theoretic proof checker ALF (the precursor of AGDA) [10, 9]. However, there the main ingredients of normalization-by-evaluation, the evaluation function and its inverse $\downarrow$ (see the end of Section 2), show up explicitely in the proofs already, while in our proof these components of the algorithm are implicit in the logical argument and are made explicit by the extraction only. There exist also formalized normalization proofs for systems such as the Calculus of Constructions [1], System F [3], the typed λ-calculus with co-products [2] and λ-calculi with various weak reduction strategies [8] for which however no program extraction by machine has been carried out.

2 The normalization proof

In this section we recall the normalization proof based on Tait's computability predicates given in [4]. The informally presented proof will serve as a template for all formalizations in this paper. At the end, we sketch the well-known normalization-by-evaluation algorithm contained in the proof.

Simple types are built from ground types ι by $\rho \Rightarrow \sigma$. The set Λ of terms is given by x^σ, $(\lambda x^\rho r^\sigma)^{\rho\Rightarrow\sigma}$, $(r^{\rho\Rightarrow\sigma} s^\rho)^\sigma$; Λ_ρ denotes the set of all terms of type ρ. *β-reduction*, $r \to s$, and the *long normal form*, $\mathrm{nf}(r)$, are defined as usual. By $r\theta$ we denote the (correct) application of a *substitution* θ to a term r and by $\theta[z \mapsto t]$ the result of overwriting θ at z to t (if θ is missing the identical substitution is to be inserted).

We will work with the following relations:

$$\begin{array}{rcl}
N_\rho(r,s) & :\Leftrightarrow & r,s \text{ are terms of type } \rho \text{ and every } \beta\text{-reduction sequence} \\
 & & r \to \ldots \text{ ends with a term having } s \text{ as its long normal form.} \\
A_\rho(r,s) & :\Leftrightarrow & r,s \text{ are terms of type } \rho \text{ and of the form } r = x r_1 \ldots r_n \\
 & & \text{and } s = x s_1 \ldots s_n \text{ with } N_{\rho_i}(r_i, s_i) \text{ for } i = 1, \ldots, n. \\
H_\rho(r,s) & :\Leftrightarrow & r,s \text{ are terms of type } \rho \text{ and of the form } r = (\lambda z^\sigma.r_1) t\,\vec{t} \\
 & & \text{and } s = r_1[z \mapsto t]\,\vec{t} \text{ where } t \text{ is } \textit{strongly normalizing}, \text{ i.e.} \\
 & & \mathrm{SN}_\sigma(t) \text{ holds (see below).} \\
F(r,k) & :\Leftrightarrow & \text{every index of a variable free in } r \text{ is } < k.
\end{array}$$

We define strong normalization, SN, and a variant, SA, by

$$\begin{array}{rcl}
\mathrm{SN}_\rho(r) & := & \forall k.F(r,k) \to \exists s N_\rho(r,s), \\
\mathrm{SA}_\rho(r) & := & \forall k.F(r,k) \to \exists s A_\rho(r,s).
\end{array}$$

The (somewhat strange looking) definition of $\mathrm{SN}_\rho(r)$ can be motivated by a computational reading: using a large enough index k we can construct a term s such that $N_\rho(r,s)$ holds. The point is that we avoid computing k from r (see Ax1 and Ax8 below).

Tait's *strong computability predicates*, SC_ρ, are defined by recursion on the type ρ:

$$\begin{array}{rcl}
\mathrm{SC}_\iota(r) & := & \mathrm{SN}_\iota(r), \\
\mathrm{SC}_{\rho\Rightarrow\sigma}(r) & := & \forall s.\mathrm{SC}_\rho(s) \to \mathrm{SC}_\sigma(rs).
\end{array}$$

The main proof, which consists of the traditional three lemmas below, will make use of the following facts:

Ax1. $F(r,k) \to N_\sigma(r x_k, s) \to N_{\rho\Rightarrow\sigma}(r, \lambda x_k^\rho s)$.

Ax2. $A_\iota(r,s) \to N_\iota(r,s)$.

Ax3. $A_\rho(x_k^\rho, x_k^\rho)$.

Ax4. $A_{\rho\Rightarrow\sigma}(r, r_1) \to N_\rho(s, s_1) \to N_\sigma(rs, r_1 s_1)$.

Ax5. $H_\rho(r, s) \to N_\rho(s, t) \to N_\rho(r, t)$.

Ax6. $\mathrm{SN}_\sigma(s) \to H_\rho((\lambda x_k^\sigma r^\rho)\theta s, r\theta[x_k \mapsto s])$.

Ax7. $H_{\rho\to\sigma}(r, s) \to H_\sigma(rt^\rho, st)$.

Ax8. $F(r, k) \to F(rx_k, k + 1)$.

Ax9. $F(rs, k) \to F(s, k)$.

Ax10. $F(rs, k) \to F(r, k)$.

Ax11. $F(r, k) \to H_\rho(r, s) \to F(s, k)$.

In the Minlog and the Isabelle/HOL formalization these facts will be taken as axioms. This does not affect program extraction because these axioms are all Harrop-formulas, i.e. have no strictly positive occurrence of an existential quantifier, and hence do not have computational content. On the other hand, in the Coq formalization Ax1-11 will be proven. Since Ax1 is not so obvious we give an informal proof (of a generalization of it):

Lemma. *If $N_\sigma(rx, s)$, where x is not free in r, then $N_{\rho\Rightarrow\sigma}(r, \lambda x^\rho s)$.*

Proof. Any reduction sequence $r \to \dots$ must be finite, since otherwise we had an infinite reduction sequence $rx \to \dots$ contradicting the assumption $N_\sigma(rx, s)$. Hence $r \to \dots \to r'$ where r' is in β-normal form. If r' is of the form $\lambda y\, t$, then $r' =_\alpha \lambda x\, t_y[x]$, since x is not free in r'. Consequently $\mathrm{nf}(r') = \lambda x\, \mathrm{nf}(t_y[x]) = \lambda x\, s$, because $rx \to^* r'x \to t_y[x]$ and $t_y[x]$ is in β-normal form. If r' is of the form $y\vec{t}$, then $\mathrm{nf}(r') = \lambda x.\mathrm{nf}(y\vec{t}x) = \lambda x\, s$, since x is not free in $y\vec{t}$ and $rx \to^* y\vec{t}x$ where $y\vec{t}x$ is β-normal. □

Lemma 1. *(a)* $\mathrm{SC}_\rho(r) \to \mathrm{SN}_\rho(r)$. *(b)* $\mathrm{SA}_\rho(r) \to \mathrm{SC}_\rho(r)$.

Proof. Induction on ρ. We will drop type indices if they are not relevant or can be inferred from the context.

Case ι. (a) holds by definition. (b). Assume $\mathrm{SA}(r)$, that is $\forall k.F(r, k) \to \exists s A(r, s)$. We must show $\mathrm{SC}_\iota(r)$, that is $\forall k.F(r, k) \to \exists s N(r, s)$. This follows from Ax2.

Case $\rho \Rightarrow \sigma$. (a). Assume $\mathrm{SC}_{\rho\Rightarrow\sigma}(r)$ and $F(r, k)$. We must show $\exists s N(r, s)$. By Ax3 we have $\mathrm{SA}(x_k)$, hence $\mathrm{SC}_\rho(k)$, by IH(b). From $\mathrm{SC}_{\rho\Rightarrow\sigma}(r)$ we conclude $\mathrm{SC}_\sigma(rx_k)$, and then $\mathrm{SN}_\sigma(rx_k)$, by IH(a). But $F(rx_k, k + 1)$, by Ax8, hence we have $N(rx_k, t)$ for some t. Now $N(r, \lambda x_k s)$, by Ax1.

(b). Assume $\mathrm{SA}_{\rho\to\sigma}(r)$ and $\mathrm{SC}_\rho(s)$. We have to show $\mathrm{SC}_\sigma(rs)$. By IH(b) it suffices to show $\mathrm{SA}(rs)$. So assume $F(rs,k)$. We must show $A(rs,t)$ for some t. We have $F(r,k)$, by Ax10. Using $\mathrm{SA}(r)$ we obtain $A(r,r_1)$ for some r_1. By Ax9 we have $F(s,k)$ and by IH(a) we know $\mathrm{SN}(s)$. Therefore $N(s,s_1)$ for some s_1. Hence $A(rs,r_1s_1)$, by Ax4. □

Lemma 2. $\mathrm{SC}_\rho(r') \to H_\rho(r,r') \to \mathrm{SC}_\rho(r)$.

Proof. Induction on ρ. **Case** ι. Assume $\mathrm{SC}_\iota(r')$, $H(r,r')$ and $F(r,k)$. We must show $N(r,s)$ for some s. Since $\mathrm{SC}_\iota(r')$ there is s s.t. $N(r',s)$, by Ax11. For the same s we have $N(r,s)$, by Ax5.

Case $\rho \Rightarrow \sigma$. Let $\mathrm{SC}_{\rho\Rightarrow\sigma}(r')$, $H(r,r')$ and assume $\mathrm{SC}_\rho(s)$. We must show $\mathrm{SC}_\sigma(rs)$. From $\mathrm{SC}_{\rho\Rightarrow\sigma}(r')$ we obtain $\mathrm{SC}_\sigma(r's)$. Hence $H(rs,r's)$, by Ax7, and $\mathrm{SC}_\sigma(rs)$, by IH. □

Lemma 3. *If r is a term of type ρ and θ a substitution such that $\mathrm{SC}_\sigma(\theta(z^\sigma))$ for all variables z^σ, then $\mathrm{SC}_\rho(r\theta)$.*

Proof. Induction on r. **Case** z^ρ. Then $\mathrm{SC}_\rho(z)$ by assumption.

Case rs. If rs has type ρ, then r must have type $\sigma \Rightarrow \rho$ for some σ with s of type σ. By IH we have $\mathrm{SC}_{\sigma\Rightarrow\rho}(r\theta)$ and $\mathrm{SC}_\sigma(s\theta)$, hence $\mathrm{SC}_\rho((rs)\theta)$.

Case $\lambda x^\sigma r$. If $\lambda x^\sigma r$ has type ρ, then $\rho = \sigma \to \tau$ with r of type τ. Assume $\mathrm{SC}_\sigma(s)$. By Lemma 1 (a), $\mathrm{SN}(s)$, hence $H(((\lambda z\, r)\theta)s, r\theta[z \mapsto \sigma])$, by Ax6. Therefore, by Lemma 2, it suffices to show $\mathrm{SC}_\tau(r\theta[z \mapsto \sigma])$, which follows from the IH for r. □

Theorem (Strong Normalization). *For every typable term r there is a term s such that $N(r,s)$.*

Proof. Let r have type ρ. By Lemma 1 (b), $\mathrm{SC}_\sigma(z)$ for every variable z^σ, by Ax3. Hence $\mathrm{SC}_\rho(r)$ by Lemma 3, hence $\mathrm{SN}(r)$ by Lemma 1 (a). Choosing a k such that $F(r,k)$ we conclude $\exists s N(r,s)$. □

A "weak" version of the normalization theorem above is obtained by replacing in the definition of the predicate N the word "every" by "there exists a", the word "ends" by "which ends" and dropping in the definition of the predicate H the premise "$\mathrm{SN}_\sigma(t)$". Then the facts Ax1-11 still hold where in Ax6 the SN-premise has to be dropped. As to be expected, the change from strong to weak normalization does not affect the extracted programs. In the first Minlog, the Coq and the Isabelle/HOL formalization we work with the (slightly simpler) weak version, while the second Minlog formalization is based on the strong version.

Normalization-by-evaluation

The algorithm contained in the proof above can be briefly described as follows. Let Λ_ρ be the (meta)type of λ-terms of (object)type ρ and define $\boldsymbol{C}_\iota := \mathrm{nat} \Rightarrow \Lambda_\iota$, $\boldsymbol{C}_{\rho\Rightarrow\sigma} := \boldsymbol{C}_\rho \Rightarrow \boldsymbol{C}_\sigma$. Define simultaneously functions $\downarrow_\rho \colon \boldsymbol{C}_\rho \Rightarrow (\mathrm{nat} \Rightarrow \Lambda_\rho)$ and $\uparrow_\rho \colon (\mathrm{nat} \Rightarrow \Lambda_\rho) \Rightarrow \boldsymbol{C}_\rho$ (called "reify" and "reflect"), by $\downarrow_\iota(g) := \uparrow_\iota(g) := g$ and

$$\begin{array}{lcl} \downarrow_{\rho\Rightarrow\sigma}(a)(k) & := & \lambda x_k^\rho \downarrow_\sigma \big(a(\uparrow_\rho(\lambda l\, x_k))\big)(k{+}1), \\ \uparrow_{\rho\Rightarrow\sigma}(g)(b) & := & \uparrow_\sigma(\lambda l.g(l)\downarrow_\rho(b)(l)). \end{array}$$

Let $[\![r^\rho]\!]_\uparrow$ denote the "value" of the term r in $\boldsymbol{C}_\rho$ under the variable assignment $\uparrow(x^\sigma) := \uparrow_\sigma(\lambda l\, x)$. Then for any k such that $F(r,k)$ holds we have

$$\mathrm{nf}(r) = \downarrow_\rho([\![r^\rho]\!]_\uparrow)(k).$$

The relations to the proof are as follows: $\downarrow$ and $\uparrow$ are the computational content of Lemma 1 (a),(b) while the evaluation function, $[\![.]\!]$, is the content of Lemma 3. The content of Lemma 2 is the identity. These relations were established in [4] in a semi formal manner and the algorithm was extended to higher-order rewrite systems in [5]. The challenge of this paper is to obtain normalization-by-evaluation fully automatically by machine-extraction.

3 Minlog formalization

Minlog (`www.minlog-system.de`) is a proof assistant intended to reason about computable functions of finite type using minimal logic. A major aim of the Minlog project is the development of practically useful tools for the machine-extraction of realistic programs from proofs. In this section we describe two Minlog formalizations of Tait's normalization proof: the weak version based on typed de Bruijn terms and the strong version based on untyped terms with named variables and a Curry-style typing.

3.1 Some background on program extraction

Before describing details of the two formalizations we provide some proof-theoretic background on program extraction and its optimization in general and on problems arising in this case study in particular.

Program extraction from constructive proofs

The method of program extraction used in this paper is based on *modified realizability* as introduced by Kreisel [12]. In short, from every constructive proof M of a non-Harrop formula A (in natural deduction or a similar proof calculus) one extracts a program $[\![M]\!]$ "realizing" A, essentially, by removing computationally irrelevant parts from the proof (proofs of Harrop formulas have no computational content). The extracted program has some simple type $\tau(A)$ which depends on the logical shape of the proven formula A only. In its original form the extraction process is fairly straightforward, but usually leads to unnecessarily complex programs. In order to obtain better programs the proof assistants in question offer various optimizations of program extraction. Below we describe such an optimization implemented in Minlog [19].

Quantifiers without computational content

Besides the usual quantifiers, $\forall$ and $\exists$, Minlog has so-called *non-computational quantifiers*, $\forall^{\mathsf{nc}}$ and $\exists^{\mathsf{nc}}$, which allow for the extraction of simpler programs. The nc-quantifiers, which were first introduced in [4], can be viewed as a refinement of the Set/Prop distinction in constructive type systems like Coq or Agda. Intuitively, a proof of $\forall^{\mathsf{nc}}xA(x)$ ($A(x)$ non-Harrop) represents a procedure that assigns to every x a proof $M(x)$ of $A(x)$ where $M(x)$ does not make "computational use" of x, i.e. the extracted program $[\![M(x)]\!]$ does not depend on x. Dually, a proof of $\exists^{\mathsf{nc}}xA(x)$ is proof of $M(x)$ for some x where the witness x is "hidden", that is, not available for computational use. Consequently, the types of extracted programs for nc-quantifiers are $\tau(\forall^{\mathsf{nc}}x^{\rho}A) = \tau(\exists^{\mathsf{nc}}x^{\rho}A) = \tau(A)$ as opposed to $\tau(\forall x^{\rho}A) = \rho \Rightarrow \tau(A)$ and $\tau(\exists x^{\rho}A) = \rho \times \tau(A)$. The extraction rules are, for example in the case of $\forall^{\mathsf{nc}}$-introduction and -elimination, $[\![(\lambda x.M^{A(x)})^{\forall^{\mathsf{nc}}xA(x)}]\!] = [\![M]\!]$ and $[\![(M^{\forall^{\mathsf{nc}}xA(x)}t)^{A(t)}]\!] = [\![M]\!]$ as opposed to $[\![(\lambda x.M^{A(x)})^{\forall xA(x)}]\!] = [\![\lambda xM]\!]$ and $[\![(M^{\forall xA(x)}t)^{A(t)}]\!] = [\![Mt]\!]$. In order for the extracted programs to be correct the variable condition for $\forall^{\mathsf{nc}}$-introduction needs to be strengthened by requiring in addition the abstracted variable x not to occur in the extracted program $[\![M]\!]$. Note that for a Harrop formula A the formulas $\forall^{\mathsf{nc}}xA$ and $\forall xA$ are equivalent, similarly, $\exists^{\mathsf{nc}}xA$ and $\exists xA$ are equivalent.

The formalization of strong computability

In the previous section we defined the formulas $\mathrm{SC}_{\rho}(r)$ by recursion on ρ. Therefore it is most natural to formally introduce SC as a (truly) dependent

family of predicates. This can and will be done in the Coq formalization in Section 4 using a definition scheme called "strong elimination". Minlog, however, being based on first-order logic, cannot define the predicate SC directly since the formulas $\mathrm{SC}_\rho(r)$ are of arbitrarily high logical complexity as ρ varies. Moreover the type $\boldsymbol{C}_\rho := \tau(\mathrm{SC}_\rho(r))$ will depend on ρ, thus a dependently typed programming language would be needed for program extraction. Our solution to this problem will be to define

$$\mathrm{SC}_\rho(r) := \exists a\, \mathrm{SCr}_\rho(a, r)$$

where SCr is a new ternary predicate which is axiomatized in such a way that $\mathrm{SCr}_\rho(a, r)$ means "a realizes $\mathrm{SC}_\rho(r)$". The type of the realizer a, call it ω, has to be equipped with constructor and accessor functions expressing that ω includes all the types $\boldsymbol{C}_\rho$. We will therefore call ω a *universal type*. Since the two Minlog formalizations will work with different kinds of typed λ-terms (Church vs. Curry) the type ω will be modeled differently as well: in the first formalization as the sum of all $\boldsymbol{C}_\rho$, in the second as the solution of a recursive type equation. Note that, as a side effect of the evasion of dependent types, *partial objects* will be needed (indicated in Minlog by a hat, $\hat{}$). An important advantage of formalizing strong computability in the way described above is the fact that the computational content of Lemma 2 in Section 2 will be extremely simple, namely the identity.

Linking realizability with truth

In order to make sure that the above definition of strong computability is equivalent to the one in Section 2 we will work in a theory where each formula is equivalent to its realizability. Normally, this is achieved by the Axioms of *Choice* and *Independence of Premises* [21]

(AC) $\quad \forall x \exists y\, A(x, y) \to \exists f \forall x\, A(x, f(x))$, where $A(x, y)$ is arbitrary,

(IP) $\quad (A \to \exists x\, B(x)) \to \exists x.A \to B(x)$, where A is a Harrop formula.

In the presence of nc-quantifiers one needs in addition a *Uniformity Principle* for the universal nc-quantifier

(UNC) $\quad \forall^{\mathsf{nc}} x \exists y\, A(x, y) \to \exists y \forall^{\mathsf{nc}} x\, A(x, y)$, where $A(x, y)$ is arbitrary.

It easy to see that all three principles are realized by identity functions.

3.2 Normalization for terms in de Bruijn notation

We now describe a Minlog formalization of the (weak) normalization proof based on typed de Bruijn terms. We first introduce a variant of de Bruijn terms [11] with typed λ-abstraction, then describe the formalization of the normalization proof, and finally discuss the extracted program (which is a collection of Minlog terms).

Simply typed terms in de Bruijn notation

Terms (in de Bruijn notation) r, s, t are built from variables n – viewed as indices – by application rs and typed abstraction $\lambda^\rho r$. A *context* is a list $\vec{\rho}$ of types. The *type* of a term r in a context $\vec{\rho}$ is defined by

$$\begin{aligned} &\mathrm{Typ}_{[]}(n) := \iota, \\ &\mathrm{Typ}_{\rho::\vec{\rho}}(0) := \rho, \\ &\mathrm{Typ}_{\rho::\vec{\rho}}(n+1) := \mathrm{Typ}_{\vec{\rho}}(n), \end{aligned} \qquad \begin{aligned} &\mathrm{Typ}_{\vec{\rho}}(rs) := \mathrm{Valtyp}(\mathrm{Typ}_{\vec{\rho}}(r)), \\ &\mathrm{Typ}_{\vec{\rho}}(\lambda^\rho r) := (\rho \Rightarrow \mathrm{Typ}_{\rho::\vec{\rho}}(r)). \end{aligned}$$

The *correctness* of a term r in a context $\vec{\rho}$ is defined by

$$\begin{aligned} \mathrm{Cor}_{\vec{\rho}}(n) &:= (n < \mathrm{Lh}(\vec{\rho})), \\ \mathrm{Cor}_{\vec{\rho}}(rs) &:= \mathrm{Cor}_{\vec{\rho}}(r) \wedge \mathrm{Cor}_{\vec{\rho}}(s) \wedge \mathrm{Typ}_{\vec{\rho}}(r) {=} (\mathrm{Typ}_{\vec{\rho}}(s) {\Rightarrow} \mathrm{Valtyp}(\mathrm{Typ}_{\vec{\rho}}(r))), \\ \mathrm{Cor}_{\vec{\rho}}(\lambda^\rho r) &:= \mathrm{Cor}_{\rho::\vec{\rho}}(r). \end{aligned}$$

The *typing judgement* $\vec{\rho} \vdash r\colon \rho$ saying that in context $\vec{\rho}$ the term r has type ρ can now be defined by

$$(\vec{\rho} \vdash r\colon \rho) := \mathrm{Cor}_{\vec{\rho}}(r) \wedge \mathrm{Typ}_{\vec{\rho}}(r) = \rho.$$

Lifting $r{\uparrow_l}$ of a term r from index l is defined by

$$n{\uparrow_l} := \begin{cases} n & \text{if } n < l \\ n+1 & \text{otherwise} \end{cases} \qquad \text{and} \qquad \begin{aligned} (rs){\uparrow_l} &:= r{\uparrow_l} s{\uparrow_l}, \\ (\lambda^\rho r){\uparrow_l} &:= \lambda^\rho r{\uparrow_{l+1}}. \end{aligned}$$

We write $r{\uparrow}$ for $r{\uparrow_0}$. *Substitution* $r[\vec{r}]$ of the first $\mathrm{Lh}(\vec{r})$ variables in a term r by the terms $\vec{r}$ is defined by

$$\begin{aligned} &n[] := n, \\ &0[r :: \vec{r}] := r, \\ &(n+1)[r :: \vec{r}] := n[\vec{r}], \end{aligned} \qquad \begin{aligned} (rs)[\vec{r}] &:= r[\vec{r}] s[\vec{r}], \\ (\lambda^\rho r)[\vec{r}] &:= \lambda^\rho r[0 :: \vec{r}{\uparrow}]. \end{aligned}$$

We let Λ denote the set of all terms and define the abstraction of the k-th variable by

$$\lambda x^\rho_k\, r := \lambda^\rho r[1, \ldots, k, 0].$$

Note that the set Λ also contains terms that do not type check.

Formalization

We begin with a description of the formal language.

Types. We have simple types (closed under products, function spaces and the list-type construction) over the base types $\mathrm{boole}, \mathrm{nat}, \mathrm{type}, \mathrm{term}, \omega$. While the base types $\mathrm{boole}, \mathrm{nat}, \mathrm{type}$ are introduced (in the proof system Minlog) as free algebras, ω is a type constant representing the infinite disjoint sum of all types $\boldsymbol{C}_\rho$ where $\boldsymbol{C}_\iota := \mathrm{nat} \Rightarrow \mathrm{term}$, $\boldsymbol{C}_{\rho\Rightarrow\sigma} := \boldsymbol{C}_\rho \Rightarrow \boldsymbol{C}_\sigma$, (see the end of Section 3.1). The type ω can be interpreted semantically by *Scott Domains*, or *Information Systems* [13].

Constants. Besides standard functions for natural numbers and lists we have function symbols for constructing, testing and manipulating types and terms (objects of type type and term). There are also "administrative" constants which should be thought as defined from the canonical injections $\mathrm{in}_\rho \colon \boldsymbol{C}_\rho \Rightarrow \omega$ and a projections $\mathrm{in}_\rho^{-1} \colon \omega \Rightarrow \boldsymbol{C}_\rho$ as follows:

$$\begin{array}{llll}
P \colon \omega \Rightarrow \mathrm{type}, & P(\mathrm{in}_\rho(u)) & := & \rho, \\
\mathrm{ModL} \colon \omega \Rightarrow \boldsymbol{N} \Rightarrow \boldsymbol{\Lambda}, & \mathrm{ModL} & := & \mathrm{in}_\iota^{-1}, \\
\mathrm{HatL} \colon \big(\boldsymbol{N} \Rightarrow \boldsymbol{\Lambda}\big) \Rightarrow \omega, & \mathrm{HatL} & := & \mathrm{in}_\iota, \\
\mathrm{Mod} \colon \omega \Rightarrow \omega \Rightarrow \omega, & \mathrm{Mod}\big(\mathrm{in}_{\rho\Rightarrow\sigma}(u)\big) & := & \mathrm{in}_\sigma \circ u \circ \mathrm{in}_\rho^{-1}, \\
\mathrm{Hat}_{\rho,\sigma} \colon \big(\omega \Rightarrow \omega\big) \Rightarrow \omega, & \mathrm{Hat}_{\rho,\sigma}(h) & := & \mathrm{in}_{\rho\Rightarrow\sigma}(\mathrm{in}_\sigma^{-1} \circ h \circ \mathrm{in}_\rho).
\end{array}$$

The functions P and Mod are undefined ($= \bot$) at arguments different to the ones shown.

Terms. Typed λ-terms over the types and constants above.

Predicate constants. Besides standard predicates, like equality at any type, we have predicate constants of the following arities ("types" is short-hand for "list type"):

$$N, A, H \colon (\mathrm{types}, \mathrm{type}, \mathrm{term}, \mathrm{term}), \qquad \begin{array}{l} F \colon (\mathrm{types}, \mathrm{type}, \mathrm{term}, \mathrm{nat}), \\ \mathrm{SCr} \colon (\mathrm{types}, \mathrm{type}, \omega, \mathrm{term}). \end{array}$$

The extra argument "types" is due to the fact that typing takes place in a context. The role of the predicate SCr was explained in Section 3.1; its meaning is expressed by the axioms below.

Formulas are built from the language above as usual, except that in addition to the ordinary quantifiers, $\forall, \exists$, we have the *nc-quantifiers*, $\forall^{\mathsf{nc}}, \exists^{\mathsf{nc}}$.

Axioms. Besides the usual logical axioms and induction axioms for the free algebras involved $(\mathrm{boole}, \mathrm{nat}, \mathrm{type}, \mathrm{term})$ we have the axioms Ax1-11 of Section 2, where typing judgements have to be added at appropriate places and the SN-hypothesis of Ax6 has to be dropped.

Furthermore, we have axioms for P, Hat and Mod,

$$\begin{aligned}
&P\big(\mathrm{Hat}_{\rho,\sigma}(h)\big) = (\rho \Rightarrow \sigma),\\
&P(a) = \iota \to \mathrm{HatL}(\mathrm{ModL}(a)) = a,\\
&\mathrm{ModL}(\mathrm{HatL}(g)) = g,\\
&P(a) = \rho \wedge P(h(a)) = \sigma \to \mathrm{Mod}\big(\mathrm{Hat}_{\rho,\sigma}(h)\big)(a) = h(a),
\end{aligned}$$

and for SCr,

$$\begin{aligned}
\mathrm{SCr}^{\iota}_{\vec\rho}(a,r) &\leftrightarrow (\vec\rho \vdash r\colon \iota) \wedge P(a) = \iota \wedge (\forall k.F^{\iota}_{\vec\rho}(r,k) \to N^{\iota}_{\vec\rho}(r,\mathrm{ModL}(a,k))),\\
\mathrm{SCr}^{\rho\Rightarrow\sigma}_{\vec\rho}(a,r) &\leftrightarrow (\vec\rho \vdash r\colon \rho\Rightarrow\sigma) \wedge P(a) = (\rho\Rightarrow\sigma) \wedge\\
&\qquad (\forall\vec\sigma, b, s.\mathrm{SCr}^{\rho}_{\vec\rho,\vec\sigma}(b,s) \to \mathrm{SCr}^{\sigma}_{\vec\rho,\vec\sigma}(\mathrm{Mod}(a,b), rs)).
\end{aligned}$$

We define SC and SN by

$$\mathrm{SC}^{\rho}_{\vec\rho}(r) := \exists a\,\mathrm{SCr}^{\rho}_{\vec\rho}(a,r), \quad \mathrm{SN}^{\rho}_{\vec\rho}(r) := \forall k.F^{\rho}_{\vec\rho}(r,k) \to \exists s N^{\rho}_{\vec\rho}(r,s).$$

The definition of SC is in harmony with the informal definition in Section 2, since, using the axioms (AC), (IP) and (UNC), one easily proves

Lemma (SC-Lemma).

$$\begin{aligned}
\forall^{\mathsf{nc}}\vec\rho, r.\,\mathrm{SC}^{\iota}_{\vec\rho}(r) &\leftrightarrow (\vec\rho \vdash r\colon \iota) \wedge \mathrm{SN}^{\iota}_{\vec\rho}(r),\\
\forall^{\mathsf{nc}}\vec\rho, r, \rho, \sigma.\,\mathrm{SC}^{\rho\Rightarrow\sigma}_{\vec\rho}(r) &\to (\vec\rho \vdash r\colon \rho\Rightarrow\sigma) \wedge (\forall^{\mathsf{nc}}\vec\sigma, s.\mathrm{SC}^{\rho}_{\vec\rho,\vec\sigma}(s) \to \mathrm{SC}^{\sigma}_{\vec\rho,\vec\sigma}(rs)).\\
\forall^{\mathsf{nc}}\vec\rho, r\forall\rho, \sigma.\,\mathrm{SC}^{\rho\Rightarrow\sigma}_{\vec\rho}(r) &\leftarrow (\vec\rho \vdash r\colon \rho\Rightarrow\sigma) \wedge (\forall^{\mathsf{nc}}\vec\sigma, s.\mathrm{SC}^{\rho}_{\vec\rho,\vec\sigma}(s) \to \mathrm{SC}^{\sigma}_{\vec\rho,\vec\sigma}(rs)).
\end{aligned}$$

The formalizations of Lemmas 1, 2, 3, and the Normalization Theorem of Section 2 read as follows:

Lemma 1.

$$\forall\rho\forall^{\mathsf{nc}}\vec\rho, r.\,(\vec\rho \vdash r\colon \rho) \to \big(\mathrm{SC}^{\rho}_{\vec\rho}(r) \to \mathrm{SN}^{\rho}_{\vec\rho}(r)\big) \wedge \big(\mathrm{SA}^{\rho}_{\vec\rho}(r) \to \mathrm{SC}^{\rho}_{\vec\rho}(r)\big).$$

Proof. Induction on ρ, as in Section 2. □

Lemma 2. $\forall^{\mathsf{nc}}\rho, \vec\rho, r, r'.\,(\vec\rho \vdash r\colon \rho) \to \mathrm{SC}^{\rho}_{\vec\rho}(r') \to H^{\rho}_{\vec\rho}(r,r') \to \mathrm{SC}^{\rho}_{\vec\rho}(r)$.

Proof. The lemma follows from the slightly stronger statement

$$\mathrm{SCr}^{\rho}_{\vec\rho}(a,r') \to H^{\rho}_{\vec\rho}(r,r') \to \mathrm{SCr}^{\rho}_{\vec\rho}(a,r),$$

which is proven by induction on ρ. Note that the induction proves a Harrop formula. Therefore the "$\forall^{\mathsf{nc}}\rho$" in the statement of the lemma is correct. □

Lemma 3. $\forall r, \vec{\rho} \forall^{\mathsf{nc}} \vec{\sigma}, \rho, \vec{s}.\, (\vec{\rho} \vdash r \colon \rho) \to \mathrm{SC}^{\vec{\rho}}_{\vec{\sigma}}(\vec{s}) \to \mathrm{SC}^{\rho}_{\vec{\sigma}}(r[\vec{s}])$.

Proof. By induction on r using Lemmas 1 and 2, as in Section 2. $\mathrm{SC}^{\vec{\rho}}_{\vec{\sigma}}(\vec{s})$ means $\mathrm{Lh}(\vec{\rho}) = \mathrm{Lh}(\vec{s}) \wedge \forall i < \mathrm{Lh}(\vec{\rho}).\, \mathrm{SC}^{\rho_i}_{\vec{\sigma}}(s_i)$. □

Theorem (Normalization). $\forall \vec{\rho}, r \forall^{\mathsf{nc}} \rho.\, (\vec{\rho} \vdash r \colon \rho) \to \exists s\, N^{\rho}_{\vec{\rho}}(r, s)$.

Proof. By Lemmas 1, 3, as in Section 2. Note that ρ can be computed from $\vec{\rho}$ and r, therefore "$\forall^{\mathsf{nc}} \rho$" is correct. □

Extracted program (Minlog)

```
Types of variables: n: nat, rho: type, r: term, a: omega
p: (omega=>nat=>term)@@((nat=>term)=>omega)
q: list type=>list omega=>omega
rhos: list type, rs: list term, as: list omega, g: nat=>term

Normalization Theorem
[rhos0,r1]
 left(cLemmaOne(Typ rhos0 r1))
 (cLemmaThree r1 rhos0(cSCrsSeq rhos0(Nil type)))
 Lh rhos0

Lemma 1
(Rec type=>(omega=>nat=>term)@@((nat=>term)=>omega))
(ModIota@([g3]OmegaInIota(cACL g3)))
([rho3,rho4,p5,p6]
  ([a7,n8]
    Abs rho3
    (Sub(left p6(Mod a7(right p5([n9]Var n8)))(Succ n8))
     ((Var map Seq 1 n8):+:(Var 0):)))@
  ([g7]
    Hat rho3 rho4
    ((cAC omega omega)
     ([a9]
       (cUNC omega)
       ((cUNC omega)((cIP omega)
        (right p6([n10]g7 n10(left p5 a9 n10)))))))))
```

The base case consists of administrative functions only. In the step case, if we disregard the administrative functions and write it out as recursion

equations, renaming according to the table

rho3	rho4	left p5	right p5	left p6	right p6
ρ	σ	$\downarrow_\rho$	$\uparrow_\rho$	$\downarrow_\sigma$	$\uparrow_\sigma$

we obtain exactly the defining equations for the functions $\downarrow$ and $\uparrow$ sketched in Section 2. From Lemma 2 we extract the identity, and from Lemma 3,

```
(Rec term=>list type=>list omega=>omega)
([n3,rhos4](ListRef omega)n3)
([r3,r4,q5,q6,rhos7,as8]Mod(q5 rhos7 as8)(q6 rhos7 as8))
([rho3,r4,q5,rhos6,as7]
  Hat rho3(Typ(rho3::rhos6)r4)
  ((cAC omega omega)
   ([a9](cUNC omega)((cUNC omega)((cIP omega)
        (q5(rho3::rhos6)(a9::as7)))))))
```

which (disregarding administrative functions) is the evaluation functional $[\![.]\!]$.

3.3 Strong normalization for terms with named variables

Now we describe the second Minlog formalization. After introducing informally untyped terms, substitutions and typing judgements we discuss the formalization and the extracted program, pointing out the main differences to the first Minlog formalization.

Simple types for untyped λ-terms

Terms r, s, t are built from variables $x, y, z, \ldots$ by application rs and abstraction $\lambda x.r$ (all untyped). A *context* is a function from variables to types. The *typing judgement* $\Gamma \vdash r : \rho$ saying that in context Γ the term r has type ρ is defined as usual. A *substitution* is a function θ from variables to terms. We will not need to formalize bound renaming since it will suffice to define the behaviour of substitution in the uncritical cases only, i.e. $y\theta = \theta(y)$, $(rs)\theta = (r\theta)(s\theta)$.

Formalization

The formal language is very similar to the first Minlog formalization. We only describe the major changes.

Types. We have the same types as in Section 3.2, however, the universal type ω will now be equipped with functions expressing that it is a solution to the recursive domain equation

$$\omega = (\text{nat} \Rightarrow \text{term}) + (\omega \Rightarrow \omega).$$

It is well-known how to solve such kind of type equations semantically. For example, a construction using information systems is described in [13].

Constants. We have the expected functions associated with the data type of untyped λ-terms, a function symbol for applying a substitution to a term and, the expected embeddings and projections for ω as a solution to the domain equation above:

$$\begin{array}{ll} \text{HatL}\colon (\text{nat} \Rightarrow \text{term}) \Rightarrow \omega, & \text{ModL}\colon \omega \Rightarrow \text{nat} \Rightarrow \text{term}, \\ \text{Hat}\colon (\omega \Rightarrow \omega) \Rightarrow \omega, & \text{Mod}\colon \omega \Rightarrow \omega \Rightarrow \omega. \end{array}$$

Predicate constants. We have again N, A, H, F and SCr, however with simpler arities, and in addition a predicate constant $\vdash$ for typing judgements:

$$\begin{array}{ll} N, A, H\colon (\text{term}, \text{term}), & \text{SCr}\colon (\omega, \text{type}, \text{term}), \\ F\colon (\text{nat}, \text{term}), & \vdash\colon (\text{nat} \Rightarrow \text{type}, \text{term}, \text{type}). \end{array}$$

However, the (informal) definition of N needs to be changed since for untyped terms the notion of a long normal form does not make sense. The new definition of the predicate N is:

$$N(r, s) \quad :\Leftrightarrow \quad \begin{array}{l} \text{every } \beta\text{-reduction sequence } r \to \ldots \text{ ends with a term} \\ \text{to which } s \ \eta\text{-reduces.} \end{array}$$

It is easy to see that the axioms (i.p. Ax1) are still valid under this new interpretation.

Axioms. The axioms Ax1-11 are as in Section 2 except that all type arguments are removed. In addition we have axioms for typing judgements:

$$(\Gamma \vdash x : \sigma) \to \sigma = \Gamma(x).$$

$$(\Gamma \vdash rs : \sigma) \to \exists^{\mathsf{nc}}\rho.(\Gamma \vdash r : \rho \Rightarrow \sigma) \wedge (\Gamma \vdash s : \rho).$$

$$(\Gamma \vdash \lambda z\, r : \rho) \to (\rho = \sigma \Rightarrow \tau) \wedge (\Gamma[z \mapsto \sigma] \vdash r : \tau).$$

The axioms concerning ω are

$$\begin{array}{l} \text{ModL}(\text{HatL}(g)) = g, \\ \text{Mod}\big(\text{Hat}(h)\big)\big) = h, \end{array}$$

and the defining axioms for SCr are

$$\begin{aligned}
&\mathrm{SCr}_\iota(a,r) &&\leftrightarrow \forall k.F(r,k) \to N(r,\mathrm{ModL}(a,k)),\\
&\mathrm{SCr}_{\rho\Rightarrow\sigma}(a,r) &&\leftrightarrow \forall b,s.\mathrm{SCr}_\rho(b,s) \to \mathrm{SCr}_\sigma(\mathrm{Mod}(a,b),rs).
\end{aligned}$$

Note that, despite the type argument ρ, the formula $\mathrm{SC}_\rho(r)$ does not include a typing statement for r. As before we define $\mathrm{SC}_\rho(r) := \exists a \mathrm{SCr}_\rho(a,r)$.

Lemma (SC-Lemma).

$$\begin{aligned}
&\forall^{\mathsf{nc}}.\mathrm{SC}_\iota(r) \;\leftrightarrow\; \mathrm{SN}(r),\\
&\forall^{\mathsf{nc}}\rho,\sigma,r.\;\; SC_{\rho\Rightarrow\sigma}(r) \leftrightarrow \forall^{\mathsf{nc}}s.\mathrm{SC}_\rho(s) \to \mathrm{SC}_\sigma(rs).
\end{aligned}$$

Below, we only list the precise formulations of the lemmas and the normalization theorem. We do not comment on proofs as they are very similar to the proofs in the previous formalization, except that contexts and substitutions are handled differently and there is less involvement of types.

Lemma 1. $\forall\rho\,\forall^{\mathsf{nc}}r.\,\big(\mathrm{SC}_\rho(r) \to \mathrm{SN}(r)\big) \wedge \big(\mathrm{SA}(r) \to \mathrm{SC}_\rho(r)\big)$.

Lemma 2. $\forall^{\mathsf{nc}}r,r',\rho.\mathrm{SC}_\rho(r') \to H(r,r') \to \mathrm{SC}_\rho(r)$.

Lemma 3. $\forall r\,\forall^{\mathsf{nc}}\Gamma,\theta,\rho.\,(\Gamma \vdash r:\rho) \to \forall z\,\mathrm{SC}_{\Gamma(z)}(\theta(z)) \to \mathrm{SC}_\rho(r\theta)$.

Theorem (Strong Normalization). $\forall r,\Gamma,\rho.(\Gamma \vdash r:\rho) \to \mathrm{SN}(r)$.

Extracted program (Minlog)

```
Types of variables: n: nat, rho: type, r: term, a: Scott(=D),
p: (Scott=>nat=>term)@@((nat=>term)=>Scott),
q: (nat=>Scott)=>Scott
rhos: nat=>type, rs: nat=>term, as: nat=>Scott

Normalization Theorem
[r0,rhos1,rho2]
 left(cLemmaOne rho2)
 (cLemmaThree r0([n4]right(cLemmaOne(rhos1 n4))([n5]Var n4)))

Lemma 1
(Rec type=>(Scott=>nat=>term)@@((nat=>term)=>Scott))
(ModL@cLemmaSCIotaFold)
([rho3,rho4,p5,p6]
  ([a7,n8]
    Abs n8(left p6(Mod a7(right p5([n9]Var n8)))(Succ n8)))@
  ([rs7]Hat(cAC([a9]right p6([n10]rs7 n10(left p5 a9 n10))))))
```

From Lemma 2 we again extract the identity, and from Lemma 3:

```
(Rec term=>(nat=>Scott)=>Scott)
([n2,as3]as3 n2)
([r2,r3,q4,q5,as6]Mod(q4 as6)(q5 as6))
([n2,r3,q4,as5]Hat(cAC([a7]q4([n8][if (n8=n2) a7 (as5 n8)])))))
```

4 Coq formalization

Coq is a proof assistant based on an higher-order type theory [20]. It includes an extraction mechanism, due originally to Christine Paulin [17, 18]. Recently, this mechanism has been deeply redesigned by the third author [14, 15], in order to remove important limitations.

4.1 Extraction in Coq

The foundations of Coq extraction are quite different from Minlog's use of Harrop formulas and nc-quantifiers. In Coq, the user decides for each new inductive datatype whether it is computationally relevant or not. For instance, the type `nat` of unary numbers is relevant, and hence placed into the universe `Set` of relevant types. On the opposite, the logical propositions `True` and `False`, defined in Coq as inductive types, are placed into the universe `Prop` of computationally irrelevant objects. This system is quite simple. In particular, when stating and proving facts, the user has no further extraction-related action to perform. The system simply verifies, via its typing rules, that the constructions of computationally relevant objects never depend on irrelevant ones.

This system is clearly not as flexible as Minlog extraction. For instance, because of its typing, a `nat` argument will always be considered relevant and kept. But objects of a same type may well be sometimes relevant and sometimes used only for filling logical annotations. For instance, considering lemma 1 and 3 above, the term argument r is relevant only in the latter. One should notice that such situations are not so common, and that the `Prop`/`Set` distinction of Coq, despite its lack of flexibility, is normally sufficient in most applications.

4.2 Higher-order logic in action

Distinctions between Coq and Minlog do not only concern extraction. During the formalization, the definition of the SC predicate has been a critical point of divergence. We have seen earlier that formalizing SC in Minlog

requires a precise understanding of the domain theory involved. Thanks to the higher-order nature of Coq logic, we can actually *define* SC:

```
Fixpoint SC (ρ:type)(r:term) {struct ρ} : Type :=
  match ρ with
   | Iota ⇒ SN Iota r
   | Arrow ρ σ ⇒∀s:term, SC ρ s → SC σ (App r s)
  end.
```

Here, we cannot define SC as an inductive predicate, since it would be non-positive and hence rejected by Coq. The alternative approach is then to see SC as a recursive function. After a case analysis on the type ρ, the two defining equations of SC are used, the second one being recursive over subterms of ρ. This kind of definition, where a new type or predicate is built via the elimination of a term such as ρ, is known as *strong elimination*. This approach is quite simple and safe: instead of using axioms for each new predicates, we rely on the correctness of Coq's typing rules.

The only delicate point with strong elimination concerns the typability of the extracted code. Using such a dependently typed construct can indeed lead to extracted code that can only be typed in a functional language with dependent types as well. Here, both before and after extraction, an object of type SC can be either a function or not, depending on the value of ρ. Since programming languages with dependent types are not widely spread, we propose some workarounds. A first solution is to use a type-free language like Scheme. Another possibility now offered by the Coq extraction is to use ML-like typing as long as possible and insert some unsafe type coercions when needed: `Obj.magic` in Ocaml and `unsafeCoerce` in Haskell. A more detailed description of this mechanism can be found in [15]. Finally, there is a last solution currently not proposed by the Coq extraction: we could here use an encoding based on a sum type like the type `D` of the Isabelle formalization (Section 5). Indeed, this solution leads to ML-typable code, but this is at the cost of additional operations (constructors and destructors like Hat and Mod in Section 3) that are useless from the operational point of view.

4.3 Organization of this formalization

The Coq formalization follow the same choices as the first Minlog formalization in Section 3.2. The first part of this work starts with some results about λ-terms in de Bruijn notation: substitution, typing, etc. Then comes

the core of the normalization proof: Lemmas 1, 2, 3 and the main theorem. This part is done in a modular way, thanks to the Coq module system. More precisely, this generic core is a *functor*, i.e. a structure that transforms a module implementing correctly predicates N, A, ... into a module containing a complete normalization proof.

Independently, we provide two implementations of these predicates and their properties. The first corresponds to the meaning of N given in Sect. 3.2, that is weak β-normalization and then η-expansion. Then, a second implementation has been done, with η-reduction instead of expansion in N, and with a small change on the abstraction function `abstr`. As noted in [4], the previous generic core can then be used unchanged, Finally, the two implementations are plugged with the core and then extracted, producing two programs which respectively compute the η-long β-normal form and the $\beta\eta$-normal form.

It should be emphasized that this formalization is complete: no axiom remains at the time of extraction. Having at least one complete implementation is crucial. For instance, proving Ax1 with de Bruijn notation has in fact been a really delicate matter, and has implied some major changes in the main part of the proof, like the use of typing contexts everywhere.

4.4 Optimizing the extracted program

The first formalized proof of normalization has been done in a direct, Coq-natural way. But the obtained extracted program, although working, was not so satisfactory. In fact, this was to be expected, since Coq extraction currently uses typing information to discover irrelevant parts in proofs. For instance, while sharing the same type `term`, the second argument r of lemma 1 is not relevant and leads to dead code after extraction, whereas the first argument r of Lemma 3 is relevant. In Minlog, the distinction between these two r arguments is done by using either the normal $\forall$ or the non-computational $\forall^{nc}$, leading to shorter and more efficient extracted code.

Recently, Bas Spitters and the third author proposed a method to emulate in Coq, at least partially, this $\forall^{nc}$ construction of Minlog [16]. The idea is to provide an injection nc that goes from the informative universe `Set` into the extraction-irrelevant universe `Prop`. Using this injection, any informative datatype like `term` has now an irrelevant counterpart `(nc term)`. The nc injection itself can be built without modifying of the Coq system. But proving properties of injected objects is currently tedious, and requires in particular the use of an technical axiom. While this axiom seems consistent and harmless for extraction, it is nevertheless unnatural in Coq, since

it contradicts the unprovable but popular principle of proof irrelevance.

Anyway, we managed to produce a second version of our formalization taking advantage of this nc construct, with only limited changes, thanks to a specialized tactic written using the rich Ltac macro language of Coq. Finally, the program extracted from this refined version is quite better. Many dead code parts are indeed removed, and some benchmarking shows a speedup of at least 40% and a memory usage decreased by more than 20%.

4.5 What Coq cannot do yet

The improved code obtained in the last section is still not perfect. Let us consider for instance the Ocaml extraction of Lemma 2.

```
let rec two t h =
  match t with
    | Iota → Obj.magic (fun k _ → Obj.magic h k __)
    | Arrow (t1, t2) →
        Obj.magic (fun _ x → two t2 (Obj.magic h __ x))
```

From an operational point of view, the type coercions `Obj.magic` could be ignored. One can then notice the use of some strange abstractions over anonymous variable `_` and applications to arbitrary constants `__`. These are the remnants of some proof abstractions and arguments. In order to support safely the whole Coq system, the new Coq extraction [15] may have to leave such remnants instead of removing them completely. Then, in an optimization phase, lots of these remnants are simplified. But the implemented optimizations currently fail to remove `_` and `__` in the above code.

In fact, in addition to this cleaning up, a more drastic optimization could be performed here. In the two Minlog formalizations, the program extracted from Lemma 2 is the identity function, due to the introduction of SC via its realizability. Here indeed, when looking closely at the code above, the result of this function is always extensionally equivalent to its second argument. However, taking advantage of this fact is currently out of the reach of Coq's extraction mechanism, which normally produces extracted terms that are faithful to the structure of the original term. Except for some local optimizations, the only difference between the original and extracted code is the pruning of logical parts, and a term built by induction like `two` will always give a recursive function. After manually implementing these remarks in the extracted code, we have noticed a new improvement of about

40% in speed and 20% in memory. The nc encoding clearly improves the efficiency of the extracted code, but remains slower than the perfect code.

More recently, we noticed that our nc encoding in Coq cannot be used in any situation. We indeed tried to switch to named λ-terms instead of de Bruijn notation, following Section 3.3. But then our nc showed its limitation. In fact, it is still based on typing information, and not on an occur-check of irrelevant variables in the extracted code like in Minlog. In particular, in the lemma 3 of Section 3.3 the simple type ρ can be considered to be without content in Minlog, while in Coq having ρ of type `(nc type)` forbids the application of ρ to the SC predicate, since it expects a first argument of type `type`. We hope to be able in the future to implement a more complete, transparent and user-friendly nc mechanism in Coq. However, this would require a modification of the system, unlike our current light nc encoding.

5 Isabelle/HOL formalization

In this section, we will compare the Coq and Minlog formalizations of the normalization proof with a formalization in the theorem prover Isabelle/HOL, which also has a program extraction facility due to the second author [7]. The formalization described here is based on the one described in Section 3.3. The LaTeX code for this section was generated by Isabelle automatically.

The datatypes of types and terms are defined as expected. The judgement stating that t has type τ in context ϱs is denoted by $\varrho s \vdash t : \tau$. The definition of the typing judgement is as usual. In contrast to Minlog, Isabelle/HOL allows the definition of predicates by recursion over datatypes (also called "strong elimination" in Coq jargon). Thus, we can simply define SC by recursion over the datatype *type*:

consts $SC :: type \Rightarrow trm \Rightarrow bool$
primrec
SC-Atom: $SC\ Iota\ r = SN\ r$
SC-Fun: $SC\ (\varrho \rightharpoonup \sigma)\ r = (\forall s.\ (SC\ \varrho\ s) \longrightarrow (SC\ \sigma\ (r \cdot s)))$

However, when it comes to extracting a program from a proof involving SC, we face a similar problem as in the Coq formalization: Since predicates in a proof become types in the extracted program, predicates such as SC defined by recursion on datatypes give rise to programs using dependent types. Such programs can neither be expressed inside Isabelle/HOL, nor can they easily be translated to functional programming languages such as ML. In order to get a program which is typable in a functional programming language without dependent types, we realize the formula $SC\ \tau\ t$ by a

datatype D with two constructors *Term* and *Func* (corresponding to the universal type ω and the functions HatL and Hat). The former corresponds to the case where τ is a base type, whereas the latter corresponds to the case where τ is a function type. In ML, one would define the datatype D as follows:

```
datatype D = Term of nat -> trm | Func of trm -> D -> D
```

This datatype is beyond the scope of Isabelle/HOL, since D occurs *negatively* in the argument type of *Func*. It is interesting to note that in the logic HOLCF, which is a conservative extension of HOL with LCF, one can actually define the above datatype, provided that the type of partial continuous functions is used instead of the type of total functions. For the moment, we just assert the existence of type D, together with suitable constructors:

typedecl D

consts
$Term :: (nat \Rightarrow trm) \Rightarrow D$
$Func :: (trm \Rightarrow D \Rightarrow D) \Rightarrow D$
$destTerm :: D \Rightarrow nat \Rightarrow trm$
$destFunc :: D \Rightarrow trm \Rightarrow D \Rightarrow D$

Here, *destTerm* and *destFunc* are *destructors* inverting the corresponding constructors of the datatype D. These can be implemented using pattern matching. We can now assign realizing terms to the two characteristic equations for SC. Since we can view an equation between propositions as a conjunction of two implications, the equations for SC are realized by pairs, of which the first component is a destructor, and the second component is a constructor:

realizers
SC-Atom: $\lambda\ t.\ (destTerm,\ Term)$
SC-Fun: $\lambda\ \varrho\ \sigma\ t.\ (destFunc,\ Func)$

The proof of strong normalization is composed of the following parts:

lemma *One* : $\forall r.\ (SC\ \varrho\ r \longrightarrow SN\ r) \wedge (SA\ r \longrightarrow SC\ \varrho\ r)$
lemma *Two* : $\forall r\ r'.\ SC\ \varrho\ r' \longrightarrow H\ r\ r' \longrightarrow SC\ \varrho\ r$
lemma *Three* : $\forall \varrho s\ \vartheta\ \varrho.\ \varrho s \vdash r : \varrho \longrightarrow (\forall z.\ SC\ (\varrho s\ z)\ (\vartheta\ z)) \longrightarrow SC\ \varrho\ (r \cdot \vartheta)$
lemma *Norm*: $\forall\ \varrho s\ \varrho\ r.\ \varrho s \vdash r : \varrho \longrightarrow (\forall k.\ F\ r\ k \longrightarrow (\exists s.\ N\ r\ s))$

The computationally relevant predicates SN and SA are defined by

$SN\ r \equiv \forall k.\ F\ r\ k \longrightarrow (\exists s.\ N\ r\ s)$
$SA\ r \equiv \forall k.\ F\ r\ k \longrightarrow (\exists s.\ A\ r\ s)$

The programs extracted from the above theorems have the types

$$
\begin{array}{ll}
\mathit{One}: & \mathit{type} \Rightarrow \mathit{trm} \Rightarrow (D \Rightarrow \mathit{nat} \Rightarrow \mathit{trm}) \times ((\mathit{nat} \Rightarrow \mathit{trm}) \Rightarrow D) \\
\mathit{Two}: & \mathit{type} \Rightarrow \mathit{trm} \Rightarrow \mathit{trm} \Rightarrow D \Rightarrow D \\
\mathit{Three}: & \mathit{trm} \Rightarrow (\mathit{name} \Rightarrow \mathit{type}) \Rightarrow (\mathit{name} \Rightarrow \mathit{trm}) \Rightarrow \\
 & \mathit{type} \Rightarrow (\mathit{name} \Rightarrow D) \Rightarrow D \\
\mathit{Norm}: & (\mathit{name} \Rightarrow \mathit{type}) \Rightarrow \mathit{type} \Rightarrow \mathit{trm} \Rightarrow \mathit{nat} \Rightarrow \mathit{trm}
\end{array}
$$

They are defined as follows:

```
One ≡
type-rec (λx. (destTerm, Term))
 (λx xa H Ha xb.
    (λHb x.
       λ[♯ x].fst (Ha (xb ¨ ´♯ x))
              (destFunc Hb (´♯ x) (snd (H (´♯ x)) (λxa. ´♯ x)))
              (Suc x) ,
     λHb. Func (λs Hc. snd (Ha (xb ¨ s)) (λx. Hb x ¨ fst (H s) Hc x))))

Two ≡
type-rec (λx xa H. Term (destTerm H))
 (λtype1 type2 H Ha r r′ Hb.
    Func (λs H. Ha (r ¨ s) (r′ ¨ s) (destFunc Hb s H)))

Three ≡
trm-rec (λname x xa xb H. H name)
 (λx xa H Ha xb xc xd Hb.
    destFunc (H xb xc (app-type-elim xb x xa xd ⇁ xd) Hb) (xa • xc)
     (Ha xb xc (app-type-elim xb x xa xd) Hb))
 (λname trm H ϱs ϑ ϱ Ha.
    let (x, y) = abs-type-elim ϱ
    in Func
       (λs Hb.
          Two y ((λ[name].trm • ϑ) ¨ s) (trm • update ϑ name s)
           (H (update ϱs name x) (update ϑ name s) y
             (λz. case name-eq-dec name z of Left ⇒ Hb
                   | Right ⇒ Ha z))))

Norm ≡
λϱs ϱ r.
  fst (One ϱ r)
   (Three r ϱs subst-id ϱ (λx. snd (One (ϱs x) (´x)) (λxa. ´x)))
```

Due to the lack of non-computational quantifiers in Isabelle, the above programs contain typing information for the term to be normalized, which is unnecessary for the computation. In particular, the extracted programs use auxiliary functions corresponding to the elimination rules

$$
\begin{array}{ll}
\textit{var-type-elim}: & \varrho s \vdash \acute{}x : \varrho \Longrightarrow \varrho = \varrho s\ x \\
\textit{abs-type-elim}: & \Gamma \vdash\ \lambda[x].t\ : \varrho \Longrightarrow \\
 & \exists \sigma\ \sigma'.\ \varrho = \sigma \rightharpoonup \sigma' \wedge\ \ \textit{update}\ \Gamma\ x\ \sigma \vdash t : \sigma' \\
\textit{app-type-elim}: & \Gamma \vdash r\ \ddot{}\ s : \varrho \Longrightarrow \exists \sigma.\ \ \Gamma \vdash r : \sigma \rightharpoonup \varrho \wedge\ \ \Gamma \vdash s : \sigma
\end{array}
$$

for the typing judgement. It turns out that all typing information can be omitted from function *Three*. This may seem a bit surprising, since *Three* calls function *Two*, which is defined by recursion on types. However, as already noticed in the Minlog and Coq previous formalizations, *Two* is actually just a complicated formulation of the identity function and can therefore be omitted. Unfortunately, Isabelle's program extraction framework cannot detect this automatically.

6 Conclusion

This case study turned out to be extremely useful for testing and comparing the proving and program extraction machineries of Minlog, Coq and Isabelle/HOL, and it led to numerous cross-fertilizations between the systems. The problems that had to be solved in connection with the representation of data, the formalization of reasoning and the optimization of program extraction pointed us to many possible ways of extending and improving the systems.

A general observation made in this case study is the fact that in order to obtain useful extracted programs, not only an optimized extraction process is needed, but also the possibility to express computational information at the logical level. For example, the virtue of the nc-quantifiers is not only their simplifying effect on the extraction process, but also their ability to make computational independencies visible in the specification and the proof.

References

[1] T. Altenkirch. Proving strong normalization of CC by modifying realizability semantics. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs. International Workshop TYPES '93. Nijmegen, The Netherlands, May 1993*, volume 806 of *LNCS*, pages 3–18. Springer Verlag, 1994.

[2] T. Altenkirch, P. Dybjer, M. Hofmann, and P. Scott. Normalization by evaluation for typed lambda calculus with coproducts. In *LICS '01:*

Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science, page 303, Washington, DC, USA, 2001. IEEE Computer Society.

[3] T. Altenkirch, M. Hofmann, and T. Streicher. Reduction-free normalisation for a polymorphic system. In *11th Annual IEEE Symposium on Logic in Computer Science*, pages 98–106, 1996.

[4] U. Berger. Program extraction from normalization proofs. In M. Bezem and J.F. Groote, editors, *Typed Lambda Calculi and Applications*, volume 664 of *LNCS*, pages 91–106. Springer Verlag, 1993.

[5] U. Berger, M. Eberl, and H. Schwichtenberg. Normalization by evaluation. In B. Möller and J.V. Tucker, editors, *Prospects for Hardware Foundations*, volume 1546 of *LNCS*, pages 117–137. Springer Verlag, 1998.

[6] U. Berger and H. Schwichtenberg. An inverse of the evaluation functional for typed λ–calculus. In R. Vemuri, editor, *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211. IEEE Computer Society Press, Los Alamitos, 1991.

[7] S. Berghofer. *Proofs, Programs and Executable Specifications in Higher Order Logic*. PhD thesis, Institut für Informatik, TU München, 2003.

[8] M. Biernacka, O. Danvy, and K. Stovring. Program extraction from proofs of weak head normalization. In *Preliminary proceedings of MFPS XXI, Birmingham, UK*, pages 105–123, 2005.

[9] C. Coquand. From semantics to rules: A machine assisted analysis. In E. Börger, Y. Gurevich, and K. Meinke, editors, *Computer Science Logic, 7th Workshop, Swansea 1993*, volume 832 of *LNCS*, pages 91–105. Springer Verlag, 1994.

[10] T. Coquand and P. Dybjer. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science*, 7:73–94, 1997.

[11] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Math.*, 34:381–392, 1972.

[12] G. Kreisel. Interpretation of analysis by means of constructive functionals of finite types. In A. Heyting, editor, *Constructivity in Mathematics*, pages 101–128. North–Holland, Amsterdam, 1959.

[13] K. G. Larsen and G. Winskel. Using information systems to solve recursive domain equations. *Information and Computation*, 91:232–258, 1991.

[14] P. Letouzey. A New Extraction for Coq. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs, Second International Workshop, TYPES 2002*, volume 2646 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[15] P. Letouzey. *Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq*. PhD thesis, Univ. Paris-Sud, 2004.

[16] P. Letouzey and B. Spitters. Implicit and noncomputational arguments using monads, 2005. Submitted for publication, available at `http://www.lri.fr/~letouzey/download/Letouzey_Spitters_05.pdf`.

[17] C. Paulin-Mohring. Extracting F_ω's programs from proofs in the Calculus of Constructions. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, Austin, January 1989. ACM Press.

[18] C. Paulin-Mohring and B. Werner. Synthesis of ML programs in the system Coq. *J. Symbolic Computation*, 11:1–34, 1993.

[19] H. Schwichtenberg. Minimal logic for computable functionals, 2004.

[20] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version 8.0*, February 2004. Available at `http://coq.inria.fr/`.

[21] A. S. Troelstra, editor. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*, volume 344 of *Lecture Notes in Mathematics*. Springer Verlag, 1973.