Getting results from programs extracted from classical proofs.*

C. Raffalli†
Université de Savoie

October 2002

Abstract

We present a new method to extract from a classical proof of $\forall x(I[x] \to \exists y(O[y] \wedge S[x,y]))$ a program computing y from x. This method applies when O is a data type and S is a decidable predicate. Algorithms extracted this way are often far better than a stupid enumeration of all the possible outputs and this is verified on a non trivial example: a proof of Dickson's lemma.

1 Introduction.

Since Griffin and Felleisen [7, 8], we know a relation between classical proofs and programs. However, it is not true that from a classical proof of the existence of an object you can compute this object. This would clearly be a contradiction with the existence of provably total but non computable functions (such as a function saying if a Turing machine will stop or not).

There are two ways of extracting a program from a classical proof:

- To associate to the absurdity rule a special control operator (the $\mathcal{C}$ operator of Felleisen also used by Krivine [11]) or the μ binder of Parigot [15, 16].

  In this case, the proof and the theorem are unchanged, but the complexity of the operator can make it difficult to understand the obtained algorithm.

- To translate the proof to intuitionistic logic using a Gödel translation or similar methods adding negations in the formula.

  In this case, it may be easier to understand the algorithm (the control operator is replaced by highly functional programs) but the theorem is changed.

Let us examine informally what happens with a classical proof of $\forall x(\mathbb{N}[x] \to \exists y(\mathbb{N}[y] \wedge S[x,y]))$ with the first approach. We will not, in general, obtain the value of y from a proof of $\mathbb{N}[x]$. Nevertheless, the algorithm will indeed build a pair whose first element is in some sense a natural number y (it is a classical proof of $\mathbb{N}[y]$) and whose second element is a classical proof p of $S[x,y]$. But, the natural y almost never satisfies the specification (this is the main difference between classical and intuitionistic proofs). However, the algorithm will backtrack and give a "better" value for y if we put together the proof p with a proof of $\neg S[x,y]$.

Nevertheless, Buchholz, Schwichtenberg and Berger in [2] were able to give a method to transform classical proofs into intuitionistic ones minimizing the number

*We wish to thank both anonymous referees for they deep comments that helped a lot in improving this paper

†email: raffalli@univ-savoie.fr

of negation. They identify a class of formulas (named "goal formulas") that do not need negation at all. In this case, if your formula $\forall x(\mathbb{N}[x] \to \exists y(\mathbb{N}[y] \wedge S[x,y]))$ is a "goal formula", then you will really get an algorithm computing y from x after transforming the classical proof.

In this paper, we present another method: we assume that $S[x,y]$ is a decidable predicate in the sense that $\forall x, y(\mathbb{N}[x] \to \mathbb{N}[y] \to (S[x,y] \vee \neg S[x,y]))$ is provable in intuitionistic logic. Then we make interact the classical proof of $\forall x(\mathbb{N}[x] \to \exists y(\mathbb{N}[y] \wedge S[x,y]))$ and the proof of decidability until we get, from x, a natural number y which really satisfies the specification.

The main result (section 5) in this paper is a proof that this interaction always terminates with a correct answer (we do not limit ourselves to the case where the input is a data type, only the output needs to be a data type).

We also test this method with a non trivial example (section 6 and 7): Dickson's lemma[1] (suggested in [2] and treated for a restricted case in [3]). This is important, because with a decidable predicate, there is always a trivial algorithm enumerating all the possible outputs and testing them until it finds the correct one. This example will show that algorithms extracted from classical proofs can be far better than this trivial algorithm.

Finally we compare our method to the method presented in [2] and [12] (section 8).

The first three sections (2 to 4) of the paper present the framework and previous results we use to state and prove our main theorem.

2 Second order mixed logic.

The main idea of our theorem is to make interact a classical proof and an intuitionistic proof. Therefore, the natural way to prove our theorem is to use mixed logic which manipulate both classical and intuitionistic reasoning.

Definition 2.1 (mixed second order formulas) *They are built from a set of intuitionistic predicate variables (infinite for each arity) and a set of classical predicate variables (also infinite for each arity). We will write the intuitionistic variables $X_i, Y_i, \ldots$ and the classical variables $X_c, Y_c, \ldots$.*

Then, from a set of first order terms $\mathcal{T}$ build over a language $\mathcal{L}$ and a set of first order variables (written $x, y, \ldots$), we build the set of formulas using the following grammar:

$$\mathcal{F} = X_i(\mathcal{T}, \ldots, \mathcal{T}) \mid X_c(\mathcal{T}, \ldots, \mathcal{T}) \mid \mathcal{F} \to \mathcal{F} \mid \forall x\mathcal{F} \mid \forall X_i\mathcal{F} \mid \forall X_c\mathcal{F}$$

Definition 2.2 (classical formulas) *A formula of mixed logic is said to be classical if its right most atomic formula is build using a classical predicate variable (regardless if it is a free or a bound predicate variable).*

Definition 2.3 (translation of formulas) *If A is a usual second order formula (using only one kind of predicate variable), We will write A_i (resp. A_c) the formula obtained by replacing all the predicate variables in A by intuitionistic (resp. classical) predicate variables with the same name.*

Definition 2.4 *In second order mixed logic, there are many ways to define the usual connectives (in fact each definition is duplicated). We define the one we use here.*

- $\bot_c := \forall X_c\, X_c$ *and* $\bot_i := \forall X_i\, X_i$

[1] We use PhoX[17] to formalize the proof and to extract and run the program.

Table 1: The rules of mixed logic

$$\frac{}{x : A, \Gamma \vdash_m x : A}\ \mathrm{Ax} \qquad \frac{\Gamma \vdash_m t : (A \to \bot_c) \to \bot_c \qquad A \text{ classical}}{\Gamma \vdash_m \mathcal{C}\, t : A}\ \mathrm{absurd}$$

$$\frac{x : A, \Gamma \vdash_m t : B}{\Gamma \vdash_m \lambda x.t : A \to B}\ \to_i \qquad \frac{\Gamma \vdash_m t : A \to B \qquad \Gamma \vdash_m u : A}{\Gamma \vdash_m t\, u : B}\ \to_e$$

$$\frac{\Gamma \vdash_m t : A \qquad x \notin \Gamma}{\Gamma \vdash_m t : \forall x\, A}\ \forall_i^1 \qquad \frac{\Gamma \vdash_m t : \forall x\, A}{\Gamma \vdash_m t : A\,[x/t]}\ \forall_e^1$$

$$\frac{\Gamma \vdash_m t : A \qquad X_i \notin \Gamma}{\Gamma \vdash_m t : \forall X_i\, A}\ \forall_i^{2i} \qquad \frac{\Gamma \vdash_m t : \forall X_i\, A}{\Gamma \vdash_m t : A\,[X_i \Leftarrow \lambda x_1, \ldots, x_n.B]}\ \forall_e^{2i}$$

$$\frac{\Gamma \vdash_m t : A \qquad X_c \notin \Gamma}{\Gamma \vdash_m t : \forall X_c\, A}\ \forall_i^{2c} \qquad \frac{\Gamma \vdash_m t : \forall X_c\, A \qquad B \text{ classical}}{\Gamma \vdash_m t : A\,[X_c \Leftarrow \lambda x_1, \ldots, x_n.B]}\ \forall_e^{2i}$$

- $\neg_c A := A \to \bot_c$ *and* $\neg_i A := A \to \bot_i$
- $A \vee_c B = \forall X_c((A \to X_c) \to (B \to X_c) \to X_c)$
- $A \vee_i B = \forall X_c((A \to X_i) \to (B \to X_i) \to X_i)$
- $A \wedge_c B = \forall X_c((A \to B \to X_c) \to X_c)$
- $A \wedge_i B = \forall X_i((A \to B \to X_i) \to X_i)$
- $\exists_c x\, A = \forall X_c(\forall x(A \to X_c) \to X_c)$ *and* $\exists_i x\, A = \forall X_i(\forall x(A \to X_i) \to X_i)$

Remark: we will also use the same definitions for usual second order predicates with only one kind of variables, for instance $A \vee B = \forall X((A \to X) \to (B \to X) \to X)$.

The rules of mixed logic and the corresponding λ-terms are given in the table 1. We use Krivine's $\lambda\mathcal{C}$-calculus [11] to interpret the algorithmic contents of the proofs. One should also say that the idea of Mixed logic is not new: Nour uses it to give a very general type to storage operators [14].

Definition 2.5 *If we restrict the rules of mixed logic to formulas using only one kind of predicate variables, we obtain the usual rules for intuitionistic or classical logic. We will write* $\Gamma \vdash_i t : A$ *for provability in intuitionistic logic and* $\Gamma \vdash_c t : A$ *for provability in classical logic.*

Lemma 2.6 *If* $\Gamma \vdash_i t : A$ *then we have* $\Gamma_i \vdash_m t : A_i$ *and* $\Gamma_c \vdash_m t : A_c$. *If* $\Gamma \vdash_c t : A$ *then we have* $\Gamma_c \vdash_m t : A_c$.

Proof : Trivial induction on the size of the proofs. ■

3 Semantics of mixed logic.

The main tool in this paper is the semantics of mixed logic. We use Krivine's "stack semantics" [10, 11].

Definition 3.1 (stacks) *A stack is a finite sequence of* λ*-terms. If* $\pi = (t_1, \ldots, t_n)$ *then* $u\,\pi$ *represents the term* $u\,t_1 \ldots t_n$.

Notation 3.2 *We write $\Lambda^{(\omega)}$ the set of all stacks.*

Definition 3.3 (weak-head reduction) *The weak-head reduction is a relation on the λ-calculus enriched with two constants $\mathcal{C}$ and $\mathcal{A}$. It is defined as the smallest relation such that:*

- $(\lambda x.t)\, u\, \pi \succ t[x/u]\, \pi$
- $\mathcal{C}\, t\, \pi \succ t\ \lambda x.(\mathcal{A}(x\, \pi))$
- $\mathcal{A}\, t\, \pi \succ t$

Remark: the weak-head reduction does not preserve types, because the second rule simplifies the term by removing the $\mathcal{C}$ combinator. We can do that when we know that the term will not receive further arguments, which is the case in the definition of the semantics below.

Definition 3.4 (stable sets) *We define $\mathcal{P}(\Lambda)^{\beta}$ (resp. $\mathcal{P}(\Lambda)^{\succ}$) the set of all sets of λ-terms closed by β-equivalence (resp. closed by weak-head expansion and β-equivalence). This means that if $\Phi \in \mathcal{P}(\Lambda)^{\succ}$, $t \in \Phi$ and if $t' \succ t$ or $t' \simeq_{\beta} t$ then $t' \in \Phi$.*

Definition 3.5 (stack duality) *If $\perp\!\!\!\perp$ is a set of λ-terms and if Π is a set of stacks, then we define $\overline{\Pi}$, the dual of Π by*

$$\overline{\Pi} = \{t \in \Lambda \mid \forall \pi \in \Pi, t\, \pi \in \perp\!\!\!\perp\}.$$

Definition 3.6 (implication) *If Φ and Ψ are sets of λ-terms, we define*

$$\Phi \to \Psi = \{t \in \Lambda \mid \forall u \in \Phi, t\, u \in \Psi\}.$$

Definition 3.7 (interpretation) *An interpretation $\mathcal{I}$ is given by*

- *an interpretation $t \mapsto |t|^{\mathcal{I}}$ from first order terms to a set $\mathcal{D}$ given by an interpretation of first order variables in $\mathcal{D}$ and an interpretation of function symbols as functions from $\mathcal{D}^n$ to $\mathcal{D}$ as usual,*
- *a set of λ-terms $\perp\!\!\!\perp_{\mathcal{I}} \in \mathcal{P}(\Lambda)^{\succ}$,*
- *for each intuitionistic predicate variable X_i of arity n a function $|X_i|^{\mathcal{I}}$ from $\mathcal{D}^n$ to $\mathcal{P}(\Lambda)^{\beta}$.*
- *for each classical predicate variable X_c of arity n a function $|X_c|^{\mathcal{I}}$ from $\mathcal{D}^n$ to $\mathcal{P}(\Lambda^{(\omega)})$. We can remark, that since $\perp\!\!\!\perp_{\mathcal{I}}$ is closed by β-equivalence, for any $(t_1, \ldots, t_n) \in \mathcal{D}^n$, $\overline{|X_c|^{\mathcal{I}}(t_1, \ldots, t_n)} \in \mathcal{P}(\Lambda)^{\beta}$.*

Remark: To simplify writing, in the notation for $\overline{\Pi}$, we omit $\perp\!\!\!\perp$. In practice, we always use this notation in a context where there is an interpretation $\mathcal{I}$, and the $\perp\!\!\!\perp$ being used in $\overline{\Pi}$ will always be $\perp\!\!\!\perp_{\mathcal{I}}$.

Remark(bis): in the definition of interpretation, we ask all the sets to be closed by β-equivalence while $\perp\!\!\!\perp_{\mathcal{I}}$ is also closed by weak-head expansion. This is fundamental to get the correctness of the semantics, because intuitively, $\perp\!\!\!\perp_{\mathcal{I}}$ is the set of programs that can really be evaluated on a machine and the reduction for $\mathcal{C}$ only apply to those terms.

Definition 3.8 *The interpretation of a formula of mixed logic is defined by induction as follows:*

- $|X_i(t_1, \ldots, t_n)|^{\mathcal{I}} = |X_i|^{\mathcal{I}}(|t_1|^{\mathcal{I}}, \ldots, |t_n|^{\mathcal{I}})$

- $|X_c(t_1,\dots,t_n)|^{\mathcal{I}} = \overline{|X_c|^{\mathcal{I}}(|t_1|^{\mathcal{I}},\dots,|t_n|^{\mathcal{I}})}$
- $|A \to B|^{\mathcal{I}} = |A|^{\mathcal{I}} \to |B|^{\mathcal{I}}$
- $|\forall x\, A|^{\mathcal{I}} = \bigcap_{u\in\mathcal{D}} |A|^{\mathcal{I}[x/u]}$
- $|\forall X_i\, A|^{\mathcal{I}} = \bigcap_{\Phi:\Lambda^n\mapsto\mathcal{P}(\Lambda)^\beta} |A|^{\mathcal{I}[X_i/\Phi]}$
- $|\forall X_c\, A|^{\mathcal{I}} = \bigcap_{\Phi:\Lambda^n\mapsto\mathcal{P}(\Lambda^{(\omega)})} |A|^{\mathcal{I}[X_c/\Phi]}$

Let $\mathcal{I}$ be an interpretation, if $\Gamma = x_1 : A_1,\dots,x_n : A_n$ is a context, we write $\sigma \in |\Gamma|^{\mathcal{I}}$ if σ is a substitution such that for all $i \in \{1,\dots,n\}$ we have $x_i[\sigma] \in |A_i|^{\mathcal{I}}$.

Lemma 3.9 *For any interpretation $\mathcal{I}$, the interpretation of a formula A belongs to $\mathcal{P}(\Lambda)^\beta$ and if A is classical, then it is the dual of a set of stacks.*

Proof : By induction on the formula A:

- The atomic case is trivial.
- For the implication case, we have: $|A \to B|^{\mathcal{I}} = \{t \in \Lambda \mid \forall u \in |A|^{\mathcal{I}}, t\,u \in |B|^{\mathcal{I}}\}$. From that fact that $|B|^{\mathcal{I}}$ is closed by β-equivalence, we know that the same is true for $|A \to B|^{\mathcal{I}}$.

  Moreover, if $A \to B$ is classical, that is if B is classical, by induction hypothesis, we know that $|B|^{\mathcal{I}} = \overline{\Pi}$ for some $\Pi \in \Lambda^{(\omega)}$. Therefore, we have $|A \to B|^{\mathcal{I}} = \overline{\Pi'}$ with $\Pi' = \{(u, t_1, \dots, t_n) \mid u \in |A|^{\mathcal{I}} \text{ and } (t_1,\dots,t_n) \in \Pi\}$.
- For the quantification case, the closure under β-reduction is trivial. For classical formulas, the result comes from the fact that $\bigcap_{i\in I} \overline{\Pi_i} = \overline{\bigcup_{i\in I} \Pi_i}$. ■

Theorem 3.10 (Correctness) *If $\Gamma \vdash_m t : A$ and if $\mathcal{I}$ is an interpretation and σ is a substitution such that $\sigma \in |\Gamma|^{\mathcal{I}}$, then $t\sigma \in |A|^{\mathcal{I}}$.*

Proof : By induction on the proof of $\Gamma \vdash_m t : A$ using the previous lemma for the case of the two elimination rules of the second order quantifiers. We will treat the only difficult case: the case of the absurdity rule:

Remark: we get easily that $|\bot_c|^{\mathcal{I}} = \{t \in \Lambda \mid \forall \pi \in \Lambda^{(\omega)}, t\,\pi \in \perp\!\!\!\perp_{\mathcal{I}}\}$ and this implies that if $t \in \perp\!\!\!\perp_{\mathcal{I}}$ then $\mathcal{A}\,t\,\pi \in |\bot_c|^{\mathcal{I}}$ (because $\mathcal{A}\,t\,\pi \succ t$ and $\perp\!\!\!\perp_{\mathcal{I}}$ is closed by weak-head expansion). Moreover, we also have $|\bot_c|^{\mathcal{I}} \subset \perp\!\!\!\perp_{\mathcal{I}}$ using the empty stack.

We choose $\sigma \in |\Gamma|^{\mathcal{I}}$ and define $t' = t\sigma$. By induction hypothesis, we have $t' \in |(A \to \bot_c) \to \bot_c|^{\mathcal{I}}$. We must prove $\mathcal{C}\,t' \in |A|^{\mathcal{I}}$. We know that A is classical and therefore, we know that $|A|^{\mathcal{I}} = \overline{\Pi} = \{u \in \Lambda \mid \forall \pi \in \Pi, u\,\pi \in \perp\!\!\!\perp_{\mathcal{I}}\}$.

Therefore, we choose $\pi \in \Pi$ and we must prove $\mathcal{C}\,t'\,\pi \in \perp\!\!\!\perp_{\mathcal{I}}$. Because $\perp\!\!\!\perp_{\mathcal{I}}$ is closed by weak-head expansion, it is enough to prove $t'\ \lambda x.(\mathcal{A}\,(x\,\pi)) \in |\bot_c|^{\mathcal{I}} \subset \perp\!\!\!\perp_{\mathcal{I}}$. To do so, using the induction hypothesis, it is sufficient to prove $\lambda x.(\mathcal{A}\,(x\,\pi)) \in |A \to \bot_c|^{\mathcal{I}}$. Then, we choose $v \in |A|^{\mathcal{I}}$ and we prove $(\lambda x.(\mathcal{A}\,(x\,\pi)))\,v \succ \mathcal{A}\,(v\,\pi) \in |\bot_c|^{\mathcal{I}}$. By the previous remark, we just have to prove $(v\,\pi) \in \perp\!\!\!\perp_{\mathcal{I}}$ which is true because $v \in |A|^{\mathcal{I}} = \overline{\Pi}$ and $\pi \in \Pi$. ■

4 Results about data types.

We give now a definition of "data types". There are many possible definitions (see [6]) and nobody knows if they are all equivalent. The one we give is a variation on these definitions, suited for our theorem. What is really important is that the data types we use verify this definition.

Definition 4.1 *An interpretation $\mathcal{I}$ is faithful for a given data type if the interpretation of all its constructors (which are function from $\mathcal{D}^n$ to $\mathcal{D}$) are injective functions with disjoint images.*

Definition 4.2 (Data types) *A data type is a second order predicate $O[x]$ with only one free first-order variable, such that*

- *O has only positive second-order quantification (it may have negative first-order quantification).*
- *For any faithful interpretation $\mathcal{I}$, if $t \in |O_i[x]|^{\mathcal{I}}$ then there exists a λ-term t' and a closed first order term u such that $t \simeq_\beta t'$, $\vdash_i t' : O[u]$ and $|x|^{\mathcal{I}} = |u|^{\mathcal{I}}$.*

Lemma 4.3 *If O is a data type, then we have the following properties:*

- *For any faithful interpretation $\mathcal{I}$, $|O_i[x]|^{\mathcal{I}} \subset |O_c[x]|^{\mathcal{I}}$. The converse is in general not true.*
- *For any faithful interpretation $\mathcal{I}$, if $t \in |O_i[x]|^{\mathcal{I}}$ then t is β-equivalent to a closed term.*

Proof : The first property is immediate because O has only positive universal quantifiers and no free second order predicate variable. The second property is a trivial consequence of the second property in the definition. ■

Proposition 4.4 *The following types are data types, if D is a data type itself (we assume that the language contains the constants 0 and nil, the unary function symbol S and the binary function symbol "::"):*

- $\mathbb{N}[x] = \forall X(X(0) \to \forall y(X(y) \to X(Sy)) \to X(x))$ *: this is the type of Church numerals.*
- $\mathbb{L}_D[l] = \forall X(X(nil) \to \forall y, n(D[n] \to X(y) \to X(n :: y)) \to X(x))$ *: this is the type of lists of elements of D.*

Proof : See [6, 9]. We will at least give the proof for natural numbers. Let $\mathcal{I}$ be a faithful interpretation for natural number. This means that the interpretation of 0 is always distinct from the interpretation of $S(x)$ and that the interpretation of S is injective.

Let us assume that $t \in |\mathbb{N}_i[x]|^{\mathcal{I}}$. Then, we choose two λ-variables a and b not free in t and we define Φ a function from $\mathcal{D}$ to $\mathcal{P}(\Lambda)^\beta$ by $\Phi(|S^n(0)|^{\mathcal{I}}) = \{t \mid t \simeq_\beta b^n a\}$ (where $b^n = b(b(b \ldots (b\, a) \ldots)))$ and $\Phi(u) = \emptyset$ otherwise.

From this and the faithfulness of $\mathcal{I}$, it is clear that $a \in |X(0)|^{\mathcal{I}[X/\Phi]}$ and $b \in |\forall y(X(y) \to X(Sy))|^{\mathcal{I}[X/\Phi]}$. From $t \in |\mathbb{N}_i[x]|^{\mathcal{I}}$, using Φ to interpret X, we deduce $t\, a\, b \in \Phi(|x|^{\mathcal{I}})$. Therefore, we have $|x|^{\mathcal{I}} = |S^n(0)|^{\mathcal{I}}$ for some $n \in \mathbb{N}$. Let $\overline{n}$ be the Church numeral for n. We have $\vdash \overline{n} : \mathbb{N}_i[n]$. And moreover, from $t\, a\, b \simeq_\beta b^n a$ we deduce easily $t \simeq_\beta \overline{n}$. ■

Proposition 4.5 *We can now type some operators we will need later:*

$$\begin{array}{ll} Z & = \lambda x.\lambda f.x \\ S & = \lambda n.\lambda x.\lambda f.(f\ (n\, x\, f)) \\ T_{\mathbb{N}} & = \lambda n.(n\ \lambda k.(k\, Z)\ \lambda g.\lambda k.(g\ (\lambda p.(k\ (S\, p))))) \\ N & = \lambda x.\lambda f.x \\ C & = \lambda a.\lambda l.\lambda x.\lambda f.(f\, a\ (l\, x\, f)) \\ T_{\mathbb{L}_{\mathbb{N}}} & = \lambda l.(l\ \lambda k.(k\, N)\ \lambda n.\lambda g.\lambda k.(T_{\mathbb{N}}\, n\, \lambda n'.(g\, \lambda l.(k\ (C\, n'\, l))))) \end{array}$$

Then we can prove in mixed logic for $O = \mathbb{N}$ or $O = \mathbb{L}_{\mathbb{N}}$ that

$$\vdash_m T_O : \forall y(O_c[y] \to \forall X_c((O_i[y] \to X_c) \to X_c))$$

Proof : Z, S, N and C are the constructors for church numerals and lists. The proof starts by instantiating the classical variable in the hypothesis $O_c[y]$ by the predicate $\lambda z.(O_i[z] \to X_c) \to X_c)$ which is indeed a classical formula. Then, the proof is easy and leads to the given terms. ■

Remark: the terms $T_{\mathbb{N}}$ and $T_{L_{\mathbb{N}}}$ are storage operators[10] respectively for Church numerals and lists. Moreover, the type given here is the most general type known for storage operators [14] but we do not use here the fact that they are storage operators, just the fact that they transform classical integers [11] into intuitionistic integers which is immediate from the correctness of the semantics.

5 The main theorem.

Theorem 5.1 *Let $I[x], O[y], S[x,y]$ be three predicates such that:*

- $\vdash_c P : \forall x(I[x] \to \exists y(O[y] \wedge S[x,y]))$
- $\vdash_i D : \forall x(I[x] \to \forall y(O[y] \to S[x,y] \vee \neg S[x,y]))$
- $\vdash_m T : \forall y(O_c[y] \to \forall X_c((O_i[y] \to X_c) \to X_c))$
- $O[y]$ *is a data type.*

Then, we define the following terms (the terms $B[i]$ and $C[i]$ depend on a parameter i, and these definitions are adjusted to avoid administrative redexes):

$$\begin{array}{lcl} A & = & \lambda o.\lambda p.o \\ B[i] & = & \lambda o.\lambda s.(D\ i\ o\ (A\ o)\ ((\lambda s'\lambda q.(q\ s'))\ s)) \\ C[i] & = & P\ i\ (\lambda e.(e\ (\lambda o'.\lambda s.(T\ o'\ \lambda o.(B[i]\ o\ s))))) \end{array}$$

If $\vdash_i i : I[u]$ then $C[i]$ will reduce to a term o such that there exists a λ-term o' and a first-order term v satisfying $o \simeq_\beta o'$, $\vdash_i o' : O[v]$ and $\vdash_i S[u,v]$ (because O is a data type, it is in general easy to compute v from o).

Remark: the term $A, B[i]$ and $C[i]$ are not typable in mixed logic with a type that would allow us to deduce our theorem directly from the adequation lemma.

How to read the term $C[i]$? Here is an informal answer (this is not a proof of the theorem, but it explains how the term $C[i]$ works).

The part $(P\ i\ (\lambda e.(e\ (\lambda o'.\lambda s \ldots$ calls the classical program P and go under the proof of the existential and the conjunction to fetch o' a classical proof of $O[y]$ and s a classical proof of $S[x,y]$. The term o' being a classical proof, the part $(T\ o'\ (\lambda o \ldots$ transforms it into an intuitionistic proof o of $O[y]$. But o may not be what we are looking for and we check by writing $(D\ i\ o\ (A\ o)\ ((\lambda s'\lambda q.(q\ s'))\ s))$ that it satisfies the specification $S[x,y]$. If this is true, $A\,o\,p$ will arrive in head position where p is a proof of $S[x,y]$ and the final result will be o. If it is false, $((\lambda s'\lambda q.(q\ s'))\ s\ q \succ q\ s$ will arrive in head position where q is a proof of $\neg S[x,y]$. Then, $(q\ s)$ is a proof of $\bot$ and the interaction of q and s will make the classical program P backtrack to propose another o' until the program terminates on a success (the program will always terminates).

Remark: We will not only prove the theorem, we will also prove that the term $C[i]$ behaves as described above. This can be expressed by the following theorem:

Theorem 5.2 *Using the hypotheses and notation of theorem 5.1, If $\vdash_i i : I[u]$ then there exists $n \in \mathbb{N}$ and two sequences of terms $o_0, \ldots, o_n$ and $s_o, \ldots, s_n$ such that:*

1. $C[i] \succ B[i]\ o_n\ s_n$
2. $B[i]\ o_n\ s_n \succ B[i]\ o_{n-1}\ s_{n-1}$

3. ...

4. $B[i]\ o_2\ s_2 \succ B[i]\ o_1\ s_1$

5. $B[i]\ o_1\ s_1 \succ A\ o_0\ s_0$

6. $A\ o_0\ s_0 \succ o_0$

7. *o_0, s_0 are β equivalent to two terms o, s such that $\vdash_i o : O[v]$ and $s_0 \in |S_i[u,v]|^{\mathcal{I}}$ for any interpretation $\mathcal{I}$ and some first order term v.*

Before proving the two theorems, we need the following lemma:

Lemma 5.3 (A property of disjunction) *If $\vdash_i t : P \vee Q$ then $t \simeq_\beta \lambda x\lambda y(x\, s_P)$ or $t \simeq_\beta \lambda x\lambda y(y\, s_Q)$ such that $s_P[x/u, x/v] \in |P_i|^{\mathcal{I}}$ and $s_P[x/u, x/v] \in |P_c|^{\mathcal{I}}$ (resp. $s_Q[x/u, x/v] \in |Q_i|^{\mathcal{I}}$ and $s_Q[x/u, x/v] \in |Q_c|^{\mathcal{I}}$) for any interpretation $\mathcal{I}$ and all terms u and v.*

Moreover, we have $\vdash_i s_P[x/u, y/u] : P$ (resp. $\vdash_i s_Q[x/u, y/u] : Q$) with $u = \lambda x\lambda y\, y$.

Proof (of the lemma): We consider t' the normal form of t (it exists and is typable because second order intuitionistic logic is strongly normalizable and satisfies subject reduction). Then, by considering the applicable rules for a normal proof of $\vdash_i t' : P \vee Q$, we get $t' = \lambda x\lambda y(x\, s_P)$ or $t' = \lambda x\lambda y(y\, s_Q)$ with in the first case $x : P \to X, y : Q \to X \vdash_i s_P : P$ and in the second case $x : P \to X, y : Q \to X \vdash_i s_Q : Q$. Moreover, the variable X is not free in P and Q.

Therefore, for any interpretation $\mathcal{I}$ and all terms u and v we have $u \in |P_i \to X|^{\mathcal{I}[\Lambda/X]}$, $v \in |Q_i \to X|^{\mathcal{I}[\Lambda/X]}$. By the adequation lemma, we have $s_P[x/u, y/v] \in |P_i|^{\mathcal{I}[\Lambda/X]} = |P_i|^{\mathcal{I}}$.

Similarly, we have $s_P[x/u, y/v] \in |P_c|^{\mathcal{I}}$, because $x : P \to X, y : Q \to X \vdash_i s_P : P$ implies $x : P \to X, y : Q \to X \vdash_c s_P : P$ and $\Lambda = \overline{\emptyset}$ is a possible classical interpretation for the variable X .

For the typability of s_P, we have $x : P \to X, y : Q \to X \vdash_i s_P : P$ implies $x : P \to I, y : Q \to I \vdash_i s_P : P$ with $I = \forall X(X \to X)$ which implies $\vdash_i s_P[x/u, y/u] : P$ because we can prove $\vdash_i u : P \to I$ and $\vdash_i u : Q \to I$ with $u = \lambda x\lambda y\, y$.

The proof for s_Q is identical. ■

Remark: from this proof, we see that it is essential that t be typable of type $I[u]$ to ensure in some sense that t belongs to both the classical and intuitionistic interpretation of $I[u]$. For the same kind of reasons, it is important that O is a data type.

Proof (of the two main theorems): We assume the hypotheses of the theorems:

- $\vdash_c P : \forall x(I[x] \to \exists y(O[y] \wedge S[x,y]))$
- $\vdash_i D : \forall x(I[x] \to \forall y(O[y] \to S[x,y] \vee \neg S[x,y]))$
- $\vdash_m T : \forall y(O_c[y] \to \forall X_c((O_i[y] \to X_c) \to X_c))$
- $\vdash_i i : I[u]$

First, we choose five distinct λ-variables $\alpha, \alpha', \delta, \gamma$ and γ' not free in P, D, T and i (easy, these are closed terms). Then, we introduce the following terms:

$$\begin{array}{rcl} C'[i] & = & P\ i\ (\lambda e.(e\ (\lambda o'.\lambda s.(T\ o'\ (\lambda o.(\delta\ o\ s))))))\\ B'[i] & = & \lambda o.\lambda s.(D\ i\ o\ (\alpha\ o)\ (\gamma\ s)) \end{array}$$

We clearly have

$$\begin{array}{lcl} B'[i][\alpha/A, \gamma/\lambda s'\lambda q.(q\ s')] & = & B[i] \\ C'[i][\delta/B'[i]][\alpha/A, \gamma/\lambda s'\lambda q.(q\ s')] & = & C'[i][\delta/B[i]] \quad = \quad C[i]. \end{array}$$

Now, we define the set $\perp\!\!\!\perp$ by induction (we choose an arbitrary interpretation $\mathcal{I}$) :

Definition 5.4 *The set $\perp\!\!\!\perp$ is the smallest set such that $t \in \perp\!\!\!\perp$ if and only if we are in one of the following cases:*

- $\exists o, s\ \ t \succ \alpha\ o\ s$ *and* $o \in |O_i[y]|^{\mathcal{I}[y/w]}$ *for some* $w \in \mathcal{D}$
- $\exists s, q\ \ t \succ \gamma\ s\ q,$ *and* $q[\alpha/\alpha', \gamma/\gamma']\ s \in \perp\!\!\!\perp$
- $\exists o, s\ \ t \succ \delta\ o\ s, B'[i]\ o\ s \in \perp\!\!\!\perp$ *and* $o \in |O_i[y]|^{\mathcal{I}[y/w]}$ *for some* $w \in \mathcal{D}$

Remark: the definition of $\perp\!\!\!\perp$ is not circular and does not depend on $\mathcal{I}$ because O_i uses only intuitionistic variables and one free first order variable. It is also clear that $\perp\!\!\!\perp$ is closed by weak-head expansion and β-equivalence and the $\perp\!\!\!\perp$ is not empty (because of the first case).

Remark: we always have $\perp\!\!\!\perp = \overline{\{\emptyset\}}$, so $\perp\!\!\!\perp$ is a possible interpretation of a classical proposition (notice that $\perp\!\!\!\perp$ is not the interpretation of $\perp_c$).

Then, we choose the interpretation $\mathcal{I}$ using the previous definition of $\perp\!\!\!\perp_{\mathcal{I}} = \perp\!\!\!\perp$ and such that $\mathcal{I}(X_c) = \perp\!\!\!\perp$.

Now the proof is decomposed in four parts:

1. We first prove $\alpha \in |\forall y(O_i[y] \to S_i[u, y] \to X_c)|^{\mathcal{I}}$. This is immediate from the definition of $\perp\!\!\!\perp$ and because $\mathcal{I}(X_c) = \perp\!\!\!\perp$.

2. Now, we prove $\delta \in |\forall y(O_i[y] \to S_c[u, y] \to X_c)|^{\mathcal{I}}$. To prove this, we assume

   $$o \in |O_i[y]|^{\mathcal{I}[y/w]}\ \ (ii) \text{ and } s \in |S_c[u, y]|^{\mathcal{I}[y/w]}\ \ (iii)$$

   and we need to prove $B'[i]\ o\ s \in \perp\!\!\!\perp$. Because $\perp\!\!\!\perp$ is closed by weak-head expansion, it is enough to prove $D\ i\ o(\alpha\ o)(\gamma\ s) \in \perp\!\!\!\perp$.

   Using the previous lemma 5.3 and (ii), we have two cases:

   - $D\ i\ o \succ \lambda x\lambda y(x\ s')$. Therefore, $D\ i\ o(\alpha\ o)(\gamma\ s) \succ \alpha\ o\ s \in \perp\!\!\!\perp$
   - $D\ i\ o \succ \lambda x\lambda y(x\ s')$ with $s'[x/t_1, y/t_2] \in |\neg_c S_c[u, y]|^{\mathcal{I}[y/w]}$ for all terms $t_1, t_2\ \ (iv)$. Therefore, $D\ i\ o(\alpha\ o)(\gamma\ s) \succ \gamma\ s\ (s'[x/\alpha\ o, y/\gamma\ s])$. We know that $s' \simeq_\beta s''$, x and y being the only free variables of s''. This implies that $s'[x/\alpha\ o, y/\gamma\ s][\alpha/\alpha', \gamma/\gamma'] \simeq_\beta s'[x/\alpha'\ o, y/\gamma'\ s]$. Then, using the definition of $\perp\!\!\!\perp$ we need to prove $s'[x/\alpha\ o, y/\gamma\ s][\alpha/\alpha', \gamma/\gamma']\ s \in \perp\!\!\!\perp$. This comes from (iii) and (iv) with $t_1 = \alpha'\ o$, $t_2 = \gamma'\ s$.

3. Next, we prove $C'[i] \in \perp\!\!\!\perp$. From the first hypothesis of the theorem, we have $P \in |\forall x(I_c[x] \to \exists_c y(O_c[y] \wedge_c S_c[x, y]))|^{\mathcal{I}}$ which implies $P\ i \in |\exists_c y(O_c[y] \wedge_c S_c[u, y]))|^{\mathcal{I}}$. This implies that $P\ i \in |\forall y(O_c[y] \wedge_c S_c[u, y] \to X_c) \to X_c|^{\mathcal{I}}$. Therefore, we just need to prove that $\lambda e.(e\ (\lambda o'.\lambda s.(T\ o'\ (\lambda o.(\delta\ o\ s)))))) \in |\forall y(O_c[y] \wedge_c S_c[u, y] \to X_c)|^{\mathcal{I}}$.To prove this, assume

   $$e \in |O_c[y] \wedge_c S_c[u, y]|^{\mathcal{I}[y/w]}$$

   and we prove $e\ (\lambda o'.\lambda s.(T\ o'\ (\lambda o.(\delta\ o\ s)))) \in \perp\!\!\!\perp$. Because of the definition of $\wedge_c$, it is enough to prove $\lambda o'.\lambda s.(T\ o'\ (\lambda o.(\delta\ o\ s))) \in |O_c[y] \to S_c[x, y] \to X_c|^{\mathcal{I}[y/w]}$. For this, we assume

   $$o' \in |O_c[y]|^{\mathcal{I}[y/w]}\ \ (v) \text{ and } s \in |S_c[u, y]|^{\mathcal{I}[y/w]}\ \ (vi)$$

and we must prove $T\ o'\ (\lambda o.(\delta\ o\ s)) \in \perp\!\!\!\perp$. But we know that $T \in |\forall y(O_c[y] \to \forall X_c((O_i[y] \to X_c) \to X_c))|^{\mathcal{I}}$. So, using (v), it is enough to prove $\lambda o.(\delta\ o\ s) \in |O_i[y] \to X_c|^{\mathcal{I}[y/w]}$ which is immediate from (2) and (vi).

4. Finally, we conclude by describing the behavior of C: From (3) we have $C'[i] \in \perp\!\!\!\perp$. Therefore, from the inductive definition of $\perp\!\!\!\perp$ and the justification below, we deduce:

   (a) $C'[i] \succ \delta\, o_n\, s_n$ with $o_n \in |O_i[y]|^{\mathcal{I}[y/w]}$ for some $w \in \mathcal{D}$

   (b) $B'[i]\, o_n\, s_n \succ \gamma\, s_n\, q_n$

   (c) $q_n[\alpha/\alpha', \gamma/\gamma']s_n \succ \delta\, o_{n-1}\, s_{n-1}$ with $o_{n-1} \in |O_i[y]|^{\mathcal{I}[y/w]}$ for some $w \in \mathcal{D}$

   (d) $B'[i]\, o_{n-1}\, s_{n-1} \succ \gamma\, s_{n-1}\, q_{n-1}$

   (e) ...

   (f) $B'[i]\, o_2\, s_2 \succ \gamma\, s_2\, q_2$

   (g) $q_2[\alpha/\alpha', \gamma/\gamma']s_2 \succ \delta\, o_1\, s_1$ with $o_1 \in |O_i[y]|^{\mathcal{I}[y/w]}$ for some $w \in \mathcal{D}$

   (h) $B'[i]\, o_1\, s_1 \succ \alpha\, o_0\, s_0$ with $o_0 \in |O_i[y]|^{\mathcal{I}[y/w]}$ for some $w \in \mathcal{D}$

   The first step is correct because α and γ are not free in $C'[i]$. Moreover, this implies that α and γ are not free in o_n nor s_n. Then, $B'[i]\, o_n\, s_n$ can either reduce to a term starting with γ (if $D\, i\, o_n \succ \lambda x\lambda y(y\, q)$) or α (if $D\, i\, o_n \succ \lambda x\lambda y(x\, q)$). This comes from the lemma 5.3 and the fact that $o_n \in |O_i[y]|^{\mathcal{I}[y/w]}$ for some $w \in \mathcal{D}$. The second case would mean we reached the last step $\alpha\, o_0\, s_0$ and $n = 1$. In the first case, we have $B'[i]\, o_n\, s_n \succ \gamma\, s_n\, q_n$. Then, $q_n[\alpha/\alpha', \gamma/\gamma']s_n$ does not contain the variable α and γ and therefore reduces to a term $\delta\, o_{n-1}\, s_{n-1}$ (because it belongs to $\perp\!\!\!\perp$). We can continue this reasoning until we reach the final step $\alpha\, o_0\, s_0$, keeping the property that α and γ never occur in o_j nor s_j for any j.

   Remark: the variables α' and γ' never arrive in head position.

   Let us define the substitution

   $$\sigma = [\alpha'/\alpha, \gamma'/\gamma][\delta/B'[i]][\alpha/A, \gamma/\lambda s'\lambda q(q\, s')].$$

   Because of the above considerations and because $C[i] = C'[i]\sigma$, we have:

   (a) $C[i] \succ B[i]\ o_n\sigma\ s_n\sigma$

   (b) $B[i]\ o_n\sigma\ s_n\sigma \succ B[i]\ o_{n-1}\sigma\ s_{n-1}\sigma$

   (c) ...

   (d) $B[i]\ o_2\sigma\ s_2\sigma \succ B[i]\ o_1\sigma\ s_1\sigma$

   (e) $B[i]\ o_1\sigma\ s_1\sigma \succ A\ o_0\sigma\ s_0\sigma$

   (f) $A\ o_0\sigma\ s_0\sigma \succ o_0\sigma$

   (g) $o_0\sigma \simeq_\beta o_0$ because o_0 is element of the data type O and therefore it is β-equivalent to a closed term.

This gives the intended behavior of C. We now have to prove that o_0 satisfies the specification.

We know that $B'[i]\, o_1\, s_1 \succ_\beta \alpha\, o_0\, s_0$. This means that $D\ i\ o_1$ reduces to $\lambda x.\lambda y.(x\, s'_0)$ (vii) and $o_0 = o_1$. This is the only possibility, because α is not free in s_1 nor o_1.

But we know that $\vdash_i i : I[u]$ and $o_0 \in |O[y]|^{\mathcal{I}[y/w]}$. From the latest we get (because O is a data type) $\vdash_i o'_0 : O[v]$ for some closed first order term v and with o'_0 the normal form of o_0.

This implies (by (vii) and lemma 5.3) that $\vdash_i s'_0[x/t, y/t] : S_i[u, v]$ for any interpretation $\mathcal{I}$ with $t = \lambda x \lambda y\, y$.

This means that $C[i] \succ o_0\sigma$ with the normal form o'_0 of $o_0\sigma \simeq_\beta o_0$ being the representation in λ-calculus of a first order term v satisfying the specification $S[u, v]$. ■

Important remark: Giving the behavior of $C[i]$, we can add to the storage operator T some "side effects" to print the term o it stores. This allows to know the sequence of values tried by the classical program before it reaches the correct answer.

6 A non trivial example: Dickson's lemma.

As suggested in [2], Dickson's lemma stated below is a good non trivial example to test program extraction from classical proofs: This lemma has a non trivial, but natural, classical proof.

Lemma 6.1 (Dickson's lemma) *For any finite sequence $(f_1, \ldots, f_n)$ of functions from $\mathbb{N}$ to $\mathbb{N}$, there is an unbounded subset X of $\mathbb{N}$ such that for all $i \in \{1, \ldots, n\}$, f_i restricted to X is an increasing function.*

Corollary 6.2 (Weak Dickson's lemma) *For any finite sequence $(f_1, \ldots, f_n)$ of functions from $\mathbb{N}$ to $\mathbb{N}$, there are arbitrary large finite subsets X of $\mathbb{N}$ such that for all $i \in \{1, \ldots, n\}$, f_i restricted to X is an increasing function.*

We will use the method described in this paper to compute these arbitrary large finite subset X of $\mathbb{N}$. First we need to give a more detailed description of the formalization, in PhoX[17], of the lemmas and some hints about the proof:

Lemma 6.3 *We define the predicate* $\mathrm{Min}[Q,\, f,\, x]$ *by*

$$\mathrm{Min}[Q,\, f,\, x] = Q(x) \wedge \forall y\, (\mathbb{N}[y] \to Q(y) \to f(x) \leq f(y))$$

which means that x is a minimum of f on the subset Q of $\mathbb{N}$. Then, we can prove the minimum principle:

$$\forall Q{\subset}\mathbb{N}\ \forall f{:}(\mathbb{N}{\mapsto}\mathbb{N})\ (\exists x\, (\mathbb{N}[x] \wedge Q(x)) \to \exists x\, (\mathbb{N}[x] \wedge \mathrm{Min}[Q,\, f,\, x]))$$

Proof : The proof uses absurdity reasoning and the well founded induction principle on $\mathbb{N}$. ■

Lemma 6.4 *Now, we define the following predicates :*

$$\begin{array}{lcl} \mathrm{Ub}[Q] & = & \forall x\, (\mathbb{N}[x] \to \exists y\, (\mathbb{N}[y] \wedge x \leq y \wedge Q(y))) \\ \mathrm{LMin}[Q,\, f,\, x] & = & Q(x) \wedge \forall y\, (\mathbb{N}[y] \to Q(y) \to x < y \to f(x) \leq f(y)) \end{array}$$

$\mathrm{Ub}[Q]$ *means that Q is unbounded and* $\mathrm{LMin}[Q,\, f,\, x]$ *means that x is a left minimum of f on Q. Then, we prove*

$$\forall Q{\subset}\mathbb{N}\ \forall f{:}(\mathbb{N}{\mapsto}\mathbb{N})\ (\mathrm{Ub}[Q] \to \mathrm{Ub}[\mathrm{LMin}(Q,f)])$$

Proof : Immediate from the previous lemma because $\mathrm{LMin}[Q,\, f,\, x]$ means $\mathrm{Min}[\lambda y(x \leq y \wedge Q(y)),\, f,\, x]$. ■

Proof (of Dickson's lemma): We can now formalize Dickson's lemma as

$$\forall l \left(\mathbb{L}_{\mathbb{N}\mapsto\mathbb{N}}[l] \to \forall Q{\subset}\mathbb{N} \left(\begin{array}{l} \mathrm{Ub}[Q] \to \\ \exists Q'{\subset}Q \left(\mathrm{Ub}[Q'] \wedge \begin{array}{l} \forall x,y{:}Q' \\ (x < y \to \forall f{\in}l\ f(x) \leq f(y)) \end{array} \right) \end{array} \right) \right)$$

which is proved by induction on the length of the list l using the previous lemma: we get an unbounded set Q such that any point of Q is a left minimum on Q for all functions in the list which is what we wanted. ■

Proof (of weak Dickson's lemma): To prove Dickson's lemma we need to manipulate finite sets of naturals. We formalize finite sets as ordered lists. Then, the specification uses the following definitions:

$$\begin{array}{lcl} \mathrm{Decreasing}[R,\ l] & & \text{means that } l \text{ is ordered for } R \text{ (defined by induction)} \\ \mathrm{R}_{l_f}[i,\ j] & = & i > j \wedge \forall f{\in}l_f\ f(i) \geq f(j) \text{ (the relation used to sort list)} \\ \mathrm{S}[l_f,\ p,\ l] & = & \mathrm{length}(l) = p \wedge \mathrm{Decreasing}[\lambda i,j\mathrm{R}_{l_f}[i,\ j],\ l]. \end{array}$$

The weak Dickson's lemma is stated as

$$\forall l_f\,(\mathbb{L}_{\mathbb{N}\mapsto\mathbb{N}}[l_f] \to \forall p\,(\mathbb{N}[p] \to \exists l\,(\mathbb{L}_{\mathbb{N}}[l] \wedge \mathrm{S}[l_f,\ p,\ l])))$$

and it is an immediate consequence of the previous lemma (by induction on the integer p) and

Proof (decidability of the specification): It is stated as follows and it is easy (but long and tedious) to prove in intuitionistic logic.

$$\forall l_f\,(\mathbb{L}_{\mathbb{N}\mapsto\mathbb{N}}[l_f] \to \forall l\,(\mathbb{N}[l] \to \forall li\,(\mathbb{L}_{\mathbb{N}}[li] \to \mathrm{S}[l_f,\ l,\ li] \vee \neg(\mathrm{S}[l_f,\ l,\ li]))))$$

■

It is clear that the hypotheses of our main theorems are verified. We have two predicates as inputs but this changes nothing. However, we really use the fact that inputs do not need to be data types because $\mathbb{L}_{\mathbb{N}\mapsto\mathbb{N}}[l_f]$ is not a data type. The output is $\mathbb{L}_{\mathbb{N}}[l]$ and we have a storage operator for it (see proposition 4.5). Finally, the specification is S and it is decidable.

There is still one problem, in the proof we use some axioms: The axioms saying that zero is distinct from successor and the successor is injective and similar axioms for lists and length of lists. These axioms are treated as follows, taking in account the need to use the classical part of the semantics (we only treat the axioms for natural numbers, the axioms for lists are treated in the same way):

Lemma 6.5 *If in the interpretation $\mathcal{I}$ we have $|0|^{\mathcal{I}} \neq |Sx|^{\mathcal{I}[x/v]}$ for any $v \in \mathcal{D}$, then we have $\lambda x.(x\,\lambda y.y) \in |\forall x(0 =_c Sx \to \bot_c)|^{\mathcal{I}}$ where $x =_c y := \forall X_c(X_c x \to X_c y)$.*

If $|S|^{\mathcal{I}}$ is an injective function, then $\lambda x.x \in |\forall x, y(Sx =_c Sy \to x =_c y)|^{\mathcal{I}}$.

Proof : For the first axiom, we assume $t \in |0 =_c Sx|^{\mathcal{I}[x/v]}$. We take Φ such that $\Phi(|0|^{\mathcal{I}}) = |\forall X_c(X_c \to X_c)|^{\mathcal{I}}$ and $\Phi(v) = |\bot_c|^{\mathcal{I}}$ if $v \neq |0|^{\mathcal{I}}$. Thus, we have $t \in |\forall X_c(X_c \to X_c)|^{\mathcal{I}} \to |\bot_c|^{\mathcal{I}}$ which implies $t\,\lambda y.y \in |\bot_c|^{\mathcal{I}}$ which is what we wanted.

For the second axiom, we assume $t \in |Sx =_c Sy|^{\mathcal{I}[x/v,y/w]}$, we must prove $t \in |x =_c y|^{\mathcal{I}[x/v,y/w]}$. To do so we choose Π from $\mathcal{D}$ to $\mathcal{P}(\Lambda^{(\omega)})$ and we must prove that $t \in \overline{\Pi(v)} \to \overline{\Pi(w)}$. From the hypothesis that $|S|^{\mathcal{I}}$ is an injective function, we choose a function p from $\mathcal{D}$ to $\mathcal{D}$ such that $p(|S|^{\mathcal{I}}(u)) = u$ for all $u \in \mathcal{D}$. Next, we take $\Pi' = \Pi \circ p$ and from the hypothesis $t \in |Sx =_c Sy|^{\mathcal{I}[x/v,y/w]}$ we get $t \in \overline{\Pi'(|S|^{\mathcal{I}}(v))} \to \overline{\Pi'(|S|^{\mathcal{I}}(w))} = \overline{\Pi(v)} \to \overline{\Pi(w)}$. ■

Remark: when we use higher-order logic, the axioms saying that constructors are distinct and injective are the only ones which are really needed. Nevertheless, we can also use equational axioms to define functions or any axiom that can be "realized" by a λ-term to preserve the adequation lemma (and the later includes the axiom of dependent choice [12] !).

Using PhoX[17], we have extracted a λ-term from the above proof and we tested it on a list of functions $f_2, \ldots, f_k$ where $f_i(p) = p \bmod i$. We give below the various

lists tried by the algorithm before it find the correct answer, for various values of k and various length n of the wanted lists. We give also the number of tries including the final correct answer.

$k = 2, n = 4$ {3 tries} {3, 2, 1, 0} {4, 3, 2, 0} {5, 4, 2, 0}

$k = 2, n = 5$ {4 tries} {4, 3, 2, 1, 0} {5, 4, 3, 2, 0} {6, 5, 4, 2, 0} {7, 6, 4, 2, 0}

$k = 3, n = 4$ {12 tries} {3, 2, 1, 0} {4, 3, 2, 0} {4, 3, 1, 0} {5, 4, 3, 0} {6, 5, 4, 0} {7, 6, 4, 0} {7, 6, 3, 0} {8, 7, 6, 0} {9, 8, 6, 0} {9, 7, 6, 0} {10, 9, 6, 0} {11, 10, 6, 0}

$k = 3, n = 5$ {19 tries} {4, 3, 2, 1, 0} {5, 4, 3, 2, 0} {5, 4, 3, 1, 0} {6, 5, 4, 3, 0} {7, 6, 5, 4, 0} {8, 7, 6, 4, 0} {8, 7, 6, 3, 0} {9, 8, 7, 6, 0} {10, 9, 8, 6, 0} {10, 9, 7, 6, 0} {11, 10, 9, 6, 0} {12, 11, 10, 6, 0} {13, 12, 10, 6, 0} {13, 12, 9, 6, 0} {14, 13, 12, 6, 0} {15, 14, 12, 6, 0} {15, 13, 12, 6, 0} {16, 15, 12, 6, 0} {17, 16, 12, 6, 0}

$k = 4, n = 4$ {26 tries} {3, 2, 1, 0} {4, 3, 2, 0} {4, 3, 1, 0} {5, 4, 3, 0} {5, 4, 2, 0} {5, 4, 1, 0} {6, 5, 4, 0} {7, 6, 4, 0} {8, 7, 6, 0} {9, 8, 6, 0} {8, 5, 4, 0} {9, 8, 4, 0} {10, 9, 4, 0} {11, 10, 9, 0} {12, 11, 10, 0} {13, 12, 10, 0} {13, 12, 9, 0} {12, 8, 4, 0} {13, 12, 4, 0} {14, 13, 12, 0} {15, 14, 12, 0} {15, 13, 12, 0} {16, 15, 12, 0} {16, 14, 12, 0} {16, 13, 12, 0} {17, 16, 12, 0}

$k = 4, n = 5$ {49 tries} {4, 3, 2, 1, 0} {5, 4, 3, 2, 0} {5, 4, 3, 1, 0} {6, 5, 4, 3, 0} {6, 5, 4, 2, 0} {6, 5, 4, 1, 0} {7, 6, 5, 4, 0} {8, 7, 6, 4, 0} {9, 8, 7, 6, 0} {10, 9, 8, 6, 0} {9, 8, 5, 4, 0} {10, 9, 8, 4, 0} {11, 10, 9, 4, 0} {12, 11, 10, 9, 0} ...

{21, 20, 16, 12, 0} {22, 21, 16, 12, 0} {23, 22, 21, 12, 0} {24, 23, 22, 12, 0} {25, 24, 22, 12, 0} {25, 24, 21, 12, 0} {24, 20, 16, 12, 0} {25, 24, 16, 12, 0} {26, 25, 24, 12, 0} {27, 26, 24, 12, 0} {27, 25, 24, 12, 0} {28, 27, 24, 12, 0} {28, 26, 24, 12, 0} {28, 25, 24, 12, 0} {29, 28, 24, 12, 0}

$k = 5, n = 4$ {114 tries} {3, 2, 1, 0} {4, 3, 2, 0} {4, 3, 1, 0} {5, 4, 3, 0} {5, 4, 2, 0} {5, 4, 1, 0} {6, 5, 4, 0} {6, 5, 3, 0} {6, 5, 2, 0} {6, 5, 1, 0} {7, 6, 5, 0} {8, 7, 6, 0} {9, 8, 6, 0} {9, 8, 5, 0} {10, 9, 8, 0} {11, 10, 9, 0}...

{50, 49, 48, 0} {51, 50, 48, 0} {50, 47, 46, 0} {51, 50, 46, 0} {51, 50, 45, 0} {52, 51, 50, 0} {52, 51, 45, 0} {53, 52, 50, 0} {53, 52, 45, 0} {53, 52, 40, 0}

$k = 5, n = 5$ {249 tries} {4, 3, 2, 1, 0} {5, 4, 3, 2, 0} {5, 4, 3, 1, 0} {6, 5, 4, 3, 0} {6, 5, 4, 2, 0} {6, 5, 4, 1, 0} {7, 6, 5, 4, 0} {7, 6, 5, 3, 0} {7, 6, 5, 2, 0} {7, 6, 5, 1, 0} {8, 7, 6, 5, 0} {9, 8, 7, 6, 0} {10, 9, 8, 6, 0} {10, 9, 8, 5, 0} ...

{110, 107, 106, 60, 0} {111, 110, 106, 60, 0} {111, 110, 105, 60, 0} {112, 111, 110, 60, 0} {112, 111, 105, 60, 0} {113, 112, 110, 60, 0} {113, 112, 105, 60, 0} {113, 112, 100, 60, 0}

It is clear that the number of tries is far below the number of subsets enumerated by a stupid algorithm. Moreover, it seems that this proof of Dickson's lemma gives the first answer for the lexicographic order (this is not at all true for all proofs).

7 Simplification of the algorithm

It is possible, by analyzing the proof (the formalization in PhoX is not very long: 343 lines including the proof of decidability), to extract (by hand) a simplified algorithm omitting parts of the proof which are not useful for the computation.

The program is written in ObjectiveCaml. The $\mathcal{C}$ operator has been removed using a CPS translation (callcc is not available in ObjectiveCaml) and the program needs recursive types to be accepted (use `ocaml -rectypes`). We now describe briefly the program.

The first function `lem1` corresponds to the lemma 6.3:

```
let lem1 f e k =
  let rec k' x q = k (x : int) ((f,k')::q) in
  e k'
```

The algorithmic content of the fact that $\mathbb{N}$ is well founded has been simplified using a recursive definition. The $\mathcal{C}$ operator has been replaced by the continuation `k`. To make the program more readable, we have stored in a list the proof that the wanted integer belongs to the desired set. The list `((f,k')::q)` is the algorithmic content of the fact that `x` is a minimum of `f` (represented by the pair `(f,k')`) and of the fact that `x` belongs to Q represented by `q`. The continuation `k'` in the pair `(f,k')` allows this function to backtrack if `x` is not a minimum of `f`. In this case, `k'` will receive in argument an integer `y` such that $f(y) < f(x)$ and such that `y` belongs to Q. Then, the function will call again the continuation `k` with `y` instead of `x`.

The lemma 6.4 (function `lem2`) is very easy and the Dickson's lemma itself (function `dickson`) does a simple induction on the list of functions:

```
let lem2 f u x = lem1 f (u (x : int))

let rec dickson lf =
  match lf with
    [] -> (fun x k -> k x [])
  | f::lf -> lem2 f (dickson lf)
```

For the weak Dickson's lemma, we have a first function `extract` receiving a proof `u` that a set Q is unbounded and building a list of elements of Q of length `n`. It is a list of pairs `(z,lz)` where `z` is the integer and `lz` is "the proof" that `z` belongs to Q. This list is passed to the continuation `k`. The weak Dickson's lemma (function `weak_dickson`) is easy using `dickson` and `extract`:

```
let rec extract n u k =
  match n with
    0 -> k []
  | x ->
    let k' l =
      match l with
        [] -> u 0 (fun z lz -> k [z,lz])
      | (y, ly)::_ -> u (y+1) (fun z lz -> k ((z,lz)::l))
    in
    extract (x - 1) u k'

let weak_dickson lf n = extract n (dickson lf)
```

Finally, the proof of decidability (function `decidable`) checks that the list satisfies the specification calling the continuation stored in the list to make the program backtrack if it is not the case. The program had to be adjusted to do the check in the same order as the term extracted from the proof.

```
let rec decidable' acc = function
    (y, ly)::l ->
      match l with
        [] -> y::acc
      | (x,lx)::_ ->
          let rec test lx' ly' =
            match lx', ly' with
              [], [] -> decidable' (y::acc) l
            | ((fx, kx)::lx'), ((fy, ky)::ly') ->
                if not (fx == fy) then
                  failwith "bug: the function should be the same !";
                if fx x < fx y then
```

```
                ky x lx' else test lx' ly'
        in test lx ly

let decidable l =
  (* side effect to print l *)
  decidable' [] (List.rev l);
```

To run the program, you can enter the following:

```
weak_dickson [(fun x -> x mod 2); (fun x -> x mod 3)] 4 decidable
```

Remark: the storage operator is not useful for this proof because the proof only constructs intuitionistic lists of naturals. This is often the case but not always (you can build examples where the storage operator will be really used to translate classical data into intuitionistic ones).

If you replace the comment "`(* side effect to print l *)`" by some code to print the integers in the list `l`, you will get exactly the same behavior (except that it is much faster) as the program extracted from the proof. This gives a strong argument to check that we did nothing wrong when simplifying the program extracted from the proof.

This program is surprisingly short and efficient and it seems that it would have been very hard to find such an algorithm directly. This means that even for a quite simple specification and a short proof, our theorem and a classical proof can lead to an interesting algorithm which is hard to construct directly.

8 Comparison with other work

comparison with the work of Berger, Buchholz and Schwichtenberg [2]

In [2] the authors have chosen a completely different approach by finding some means to transform the classical proof in an intuitionistic one. They do not use a Gödel translation. They use a logical framework with only one decidable predicate symbol and only the connectives $\bot, \rightarrow, \forall$ which implies that classical and intuitionistic logic are equivalent in this setting.

However, to code the formula you want to prove, you need to add negations (at least two negations for the existential quantifier because the formula you want to prove is $\forall x \exists y G[x, y]$).

Then, the authors prove that if G is a so called "goal formula" and if you prove $\vdash \forall x(\forall y(G[x, y] \rightarrow \bot) \rightarrow \bot)$ then you can prove $\vdash \forall x G[x, Mx]$ where M is a functional program computing y from x.

We could consider that goal formulas play a similar role as decidable predicates, but what is the relation between these two notions ? Goal formulas do not necessarily define decidable predicates (they may contain universal quantifications), but in this case the method in [2] will probably not lead to an algorithm ?).

A first clue is that, in our method, if we change the decidability proof, it changes the order in which we check that the solution is correct and the program behaves differently. In [2] it seems that lemma 3.1 plays a similar role. This lemma says that for any goal formula G, we can prove that $\vdash G^X[x, y] \rightarrow (G[x, y] \rightarrow X) \rightarrow X$ where $G^X[x, y]$ is the formula $G[x, y]$ where all occurrences of $\bot$ have been replaced by a variable X. It is clear that there may be many proofs of $\vdash G^X[x, y] \rightarrow (G[x, y] \rightarrow X) \rightarrow X$ and this probably introduces the same kind of variation in the behavior of the extracted algorithm.

However, we do not see a clear relation between the proof of $\vdash G^X[x, y] \rightarrow (G[x, y] \rightarrow X) \rightarrow X$ and a proof of decidability of the predicate $G[x, y]$. Yet, in

[3], using the method in [2], a program is extracted from a restricted version of Dickson's lemma (only two functions and two natural numbers) and is very similar to ours...

comparison with [12]

In this paper, Krivine shows how the λC-term extracted from a classical proof can be directly used as a winning strategy for Coquand's game semantics [4]. Krivine studies formulas written $\forall x_1{\in}\mathbb{N}\exists y_1{\in}\mathbb{N}\ldots\forall x_n{\in}\mathbb{N}\exists y_n{\in}\mathbb{N}\ \Phi(x_1,y_1,\ldots,x_n,y_n)=0$ where Φ is a recursive function.

Therefore, the case $n=1$ ($\forall x{\in}\mathbb{N}\exists y{\in}\mathbb{N}\,\Phi(x,y)=0$) is a particular case of our work because the equality is decidable and it is very easy to make the λC-term extracted from a classical proof computes y from x (intuitively, because the decidability proof is trivial).

What do we gain by using formulas of the shape $\forall x{\in}\mathbb{N}\exists y{\in}\mathbb{N}\,S(x,y)$ instead of $\forall x{\in}\mathbb{N}\exists y{\in}\mathbb{N}\,\Phi(x,y)=0$? First it is easy to go from one to another (the second case is a particular case of the first one and from a decidable predicate we can easily get a recursive function). However, it would be very difficult to analyze the behavior of the classical proof of Dickson's lemma (or any large proof), because the decidability proof (which has an obvious algorithmic contents) is replaced by a proof that $\forall x{\in}\mathbb{N}\forall y{\in}\mathbb{N}\,(S(x,y)\leftrightarrow(\Phi(x,y)=0))$. Moreover, in our paper the program is a well analyzed interaction between two components: the intuitionistic decidability proof and the classical existence proof, which helps a lot in understanding the whole program (this is the main point of our theorem). In Krivine's work, there is only one monolithic proof to analyze.

One should note that Krivine's work is more general because proofs can use the axiom of dependent choice and the alternation of quantifiers is not limited (but in this case we can not anymore compute the value of y_1 from x_1).

It is clear that our work can be extended to use the axiom of dependent choice because the justification uses the same semantics and there is only to prove that the term chosen for the axiom of choice preserves the adequation lemma which is the case.

We also think that Krivine's work can be extended to formula of the shape $\forall x_1{\in}\mathbb{N}\exists y_1{\in}\mathbb{N}\ldots\forall x_n{\in}\mathbb{N}\exists y_n{\in}\mathbb{N}\ S(x_1,y_1,\ldots,x_n,y_n)$ where S is a decidable predicate using our technics, but this needs to be checked.

9 Further works

There is at least two open questions:

1. What is the relation between goal formulas and decidable formulas and more generally, can we prove a syntactical result similar to our theorem:

   Conjecture 9.1 *If S is decidable and if O is a data type (for a well chosen definition), then from a classical proof of $\forall x(I[x]\to\exists y(O[y]\wedge S[x,y]))$ we can build an intuitionistic one (it would be better not just to find a proof, but really give a constructive transformation of the classical proof into an intuitionistic one).*

   This conjecture would be a consequence of the existence of typed storage operators for all decidable predicates. Which is equivalent to the following question: "Can we prove $\forall x(I(x)\to\forall y(O[y]\to S^*[x,y]\to\neg\neg S[x,y])$ where S^* is a Gödel translation of S ?". This would allow a proof similar to the proof

that all functions provably total in classical second order logic are provably total in intuitionistic logic.

2. There is a huge difference between the extracted term and the simplified program. Can we reduce these differences automatically using methods similar to those in [1, 5, 13] developed for intuitionistic logic. However, with classical logic, there is one added problem compared to intuitionistic logic: the equational reasoning can have an algorithmic content: the equality proofs can trigger the backtracking of the program.

References

[1] Stefano Berardi. Pruning simply typed λ-terms. *Journal of Logic and Computation*, 6:663–681, 1996.

[2] Ulrich Berger, Wilfried Buchholz, and Helmut Schwichtenberg. Refined program extraction from classical proofs. *Annals of Pure and Applied Logic*, 114:3–25, 2002.

[3] Ulrich Berger, Helmut Schwichtenberg, and Seisenberger Monika. The Warshall Algorithm and Dickson's Lemma: Two Examples of Realistic Program Extraction. *Journal of Automated Reasoning*, 26, 2001.

[4] Thierry Coquand. A semantics of evidence for classical arithmetic. *Journal of Symbolic Logic*, 60:325–337, 1995.

[5] Philippe Curmin. *Marquage des preuves et extraction de programmes*. PhD thesis, Équipe de logique Université Paris VII, 1999.

[6] Samir Farkh. *Types de données en logique du second ordre*. PhD thesis, Université de Savoie, Décembre 1998. Directeur: R. David et K. Nour.

[7] M. Felleisen and D. Friedman. Control operators, the SECD machine and the λ-calculus. *Formal description of Programming Concepts III*, pages 131–141, 1986.

[8] Timothy G. Griffin. A formulae as-as-types notion of control. In *17th annual ACM symposiuom on Principle of Programming Language*, pages 47–58. Oxford University Press, 1990.

[9] Jean-Louis Krivine. *Lambda-Calculus : Types and Models*. Computers and their applications. Ellis Horwood (or Masson in French), 1993.

[10] Jean-Louis Krivine. A general storage theorem for integers in call-by-name lambda-calculus. *Theoretical Computer Science*, 129:79–94, 1994.

[11] Jean-Louis Krivine. Classical logic, storage operators and second-order lambda-calculus. *Annals of Pure and Applied Logic*, 68:225–260, 1994.

[12] Jean-Louis Krivine. Dependent choice, 'quote' and the clock. *Th. Comp. Sc.*, 308:259–276, 2003.

[13] Boerio L. *Optimizing Programs Extracted From Proofs*. PhD thesis, Computer Science Department of Torino University, 1995.

[14] Karim Nour. Mixed logic and storage operators. *Archive for Mathematical Logic*, 39:261–280, 2000.

[15] Michel Parigot. $\lambda\mu$-calculus an algorithmic interpretation of classical natural deduction. *Proceedings of Logic and Automatic Reasoning*, 1991. Lecture Notes in Computer Science Vol. 624.

[16] Michel Parigot. Strong Normalization for Second Order Classical Natural Deduction. In *Logic in Computer Sciences*, pages 39–46, 1993.

[17] Christophe Raffalli. "the phox proof assistant version 0.8". software available on the Internet: `http://www.lama.univ-savoie.fr/~RAFFALLI/phox.html`, 2002.

A The λ-terms

Here are the real λ-terms extracted from the proof except

- the first height which are axioms,
- the final term `interact` which corresponds to $\lambda i C[i]$ (in our example we have two arguments in input: the list of functions and the number of integers we want, this is why the term starts with $\lambda l \lambda n \ldots$)
- and the terms `PL` and `PN` that are used to print the various lists tested by the decidability proof while behaving like the identity on the intended data type).

From our web page `www.lama.univ-savoie.fr/~raffalli`, you can download the three files `dickson.phx`, `storage.phx` and `run_dickson.phx` to run yourself the real extracted program using PhoX. To do so you need to compile the first two files using the command `phox -c filename.phx` then, you can run the last file using the command `phox -f < filename.phx`.

```
peirce_law = \x0 a1 [a1]x0 (\x2 [a1]x2)
S_inj.N = \x0 \x1 \x2 x2
N0_not_S.N = \x0 \x1 x1
equal.reflexive = \x0 x0
cons.injective.List = \x0 \x1 x1 x0 x0
nil_not_cons.List = \x0 x0
length.nil.List = \x0 x0
length.cons.List = \x0 \x1 x1
lesseq.refl.N = \x0 \x1 \x2 x1
conjunction.intro = \x0 \x1 \x2 x2 x0 x1
exists.intro = \x0 \x1 x1 x0
exists.elim = \x0 \x1 x1 x0
conjunction.left = \x0 \x1 x1 x0
S.total.N = \x0 \x1 \x2 x2 (x0 x1 x2)
disjunction.right.intro = \x0 \x1 \x2 x2 x0
N0.total.N = \x0 \x1 x0
disjunction.elim = \x0 \x1 \x2 x2 x0 x1
disjunction.left.intro = \x0 \x1 \x2 x1 x0
case.N = \x0 x0 (disjunction.left.intro equal.reflexive)
  (\x1 disjunction.elim
    (\x2 disjunction.right.intro
      (exists.intro
        (conjunction.intro (x2 (\x3 x3) N0.total.N)
          equal.reflexive)))
    (\x2 exists.elim
      (\x3 conjunction.left
        (\x4 \x5 disjunction.right.intro
          (exists.intro
            (conjunction.intro (x5 (\x6 x6) (S.total.N x4))
              equal.reflexive))) x3) x2) x1)
case_left.N = \x0 \x1 \x2 disjunction.elim
  (\x3 x3 (\x4 x4) (x0 x3))
  (\x3 exists.elim
    (\x4 conjunction.left (\x5 \x6 x6 (\x7 x7) (x1 x5 x6)) x4) x3)
  (case.N x2)
lesseq.lS.N = \x0 \x1 \x2 \x3 x1 x2 (\x4 x3 x4)
conjunction.right.elim = \x0 x0 (\x1 \x2 x2)
rec.N = \x0 \x1 \x2 conjunction.right.elim
  (x2 (conjunction.intro N0.total.N x0)
    (\x3 conjunction.left
      (\x4 \x5 conjunction.intro (S.total.N x4) (x1 x4 x5)) x3))
lesseq.lN0.N = \x0 \x1 \x2 rec.N x1 (\x3 \x4 x2 x4) x0
false.elim = \x0 x0
not.lesseq.imply.less.N = \x0 rec.N
  (\x1 \x2 false.elim (x2 (lesseq.lN0.N x1)))
  (\x1 \x2 \x3 \x4 disjunction.elim
    (\x5 lesseq.lS.N x3 (x5 (\x6 x6) (lesseq.lN0.N x1)))
    (\x5 exists.elim
      (\x6 conjunction.left
        (\x7 \x8 lesseq.lS.N x3
          (x8 (\x9 x9)
            (x2 x7 (\x9 x4 (x8 (\x10 x10) (lesseq.lS.N x1 x9))))))
        x6) x5) (case.N x3)) x0
absurd = \x0 peirce_law x0
lesseq.rS.N = \x0 \x1 \x2 \x3 x3 (x1 x2 x3)
lesseq.rec.N = \x0 \x1 \x2 \x3 \x4 conjunction.left
  (\x5 \x6 conjunction.left (\x7 \x8 x8) x6)
  (x4
    (conjunction.intro (lesseq.refl.N x0)
      (conjunction.intro x0 x2))
    (\x5 conjunction.left
      (\x6 \x7 conjunction.left
        (\x8 \x9 conjunction.intro (lesseq.rS.N x0 x6)
          (conjunction.intro (S.total.N x8)
            (case_left.N (\x10 x3 (x10 x8) (x10 x6) (x10 x9))
              (\x10 \x11 x3 (x11 x8) (x11 x6) (x11 x9)) x8))) x7)
      x5))
S_inj_left.N = \x0 \x1 \x2 \x3 x2 (S_inj.N x0 x1 x3)
lesseq.S_inj.N = \x0 \x1 \x2 exists.elim
  (\x3 conjunction.left
    (\x4 \x5 conjunction.left
      (\x6 \x7 S_inj_left.N x1 x4 (\x8 x8 (\x9 x9) x7) x6) x5) x3)
  (lesseq.rec.N (S.total.N x0) (S.total.N x1)
    (exists.intro
      (conjunction.intro x0
        (conjunction.intro equal.reflexive (lesseq.refl.N x0))))
    (\x3 \x4 \x5 exists.elim
      (\x6 conjunction.left
        (\x7 \x8 conjunction.left
          (\x9 \x10 exists.intro
            (conjunction.intro (x9 x3)
              (conjunction.intro (x9 equal.reflexive)
                (lesseq.rS.N x0 x10)))) x8) x6) x5) x2)
lesseq.S_inj_left.N = \x0 \x1 \x2 \x3 x2 (lesseq.S_inj.N x0 x1 x3)
lesseq.or_eq_S.N = \x0 \x1 case_left.N
  (\x2 \x3 disjunction.left.intro (lesseq.lN0.N x1))
  (\x2 \x3 \x4 lesseq.S_inj_left.N x2 x1
    (\x5 lesseq.rec.N x2 x1
      (disjunction.right.intro equal.reflexive)
      (\x6 \x7 \x8 disjunction.left.intro (lesseq.lS.N x2 x7)) x5)
    x4) x0
lesseq.or_eq_S_left.N = \x0 \x1 \x2 \x3 \x4 disjunction.elim
  (\x5 x2 x5) (\x5 x3 x5) (lesseq.or_eq_S.N x0 x1 x4)
S_not_N0.N = \x0 \x1 N0_not_S.N x0 (x1 (\x2 x2) equal.reflexive)
lesseq.rN0.N = \x0 \x1 case_left.N (\x2 x2 x2)
  (\x2 \x3 false.elim
    (lesseq.rec.N x0 N0.total.N (\x4 N0_not_S.N x2 (x3 x4))
      (\x4 \x5 \x6 \x7 N0_not_S.N x4 x7) x1 equal.reflexive)) x0
```

```
lesseq.rN0_left.N = \x0 \x1 \x2 x1 (lesseq.rN0.N x0 x2)
lesseq.rN1.N = \x0 \x1 lesseq.rN0_left.N (S.total.N x0)
  (\x2 S_not_N0.N x0 x2) x1
well_founded.N = \x0 \x1 rec.N
  (\x2 \x3 lesseq.rN0_left.N x2
    (\x4 x0 x2 (\x5 \x6 lesseq.rN1.N x5 (x4 x6))) x3)
  (\x2 \x3 \x4 \x5 lesseq.or_eq_S_left.N x4 x2
    (\x6 case_left.N (\x7 x3 (x7 x4) (x7 x6))
      (\x7 \x8 x3 (x8 x4) (x8 x6)) x4)
    (\x6 x0 x4
      (\x7 \x8 lesseq.S_inj_left.N x7 x2 (\x9 x3 x7 x9) (x6 x8)))
    x5) x1 x1 (lesseq.refl.N x1)
lem1 = \x0 \x1 \x2 exists.elim
  (\x3 conjunction.left
    (\x4 \x5 absurd
      (\x6 false.elim
        ((\x7 well_founded.N
           (\x8 \x9 \x10 \x11 \x12 exists.elim
             (\x13 conjunction.left
               (\x14 \x15 conjunction.left
                 (\x16 \x17 x9
                   (case_left.N (\x18 x1 (x18 x14))
                     (\x18 \x19 x1 (x19 x14)) x14)
                   (x11 (\x18 x18) x17) x14 equal.reflexive x16)
                 x15) x13)
             ((\x13 \x14 absurd
                (\x15 false.elim
                  (x6
                    (exists.intro
                      (conjunction.intro x13
                        (conjunction.intro x14
                          (\x16 \x17 absurd
                            (\x18 false.elim
                              (x15
                                (exists.intro
                                  (conjunction.intro x16
                                    (conjunction.intro x17
                                      (not.lesseq.imply.less.N
                                        (case_left.N
                                          (\x19 x1 (x19 x13))
                                          (\x19 \x20 x1 (x20 x13))
                                          x13)
                                        (case_left.N
                                          (\x19 x1 (x19 x16))
                                          (\x19 \x20 x1 (x20 x16))
                                          x16) x18)))))))))))))
           x10 x12)) x7) (x1 x4) x4 equal.reflexive x5))) x3)
  x2
lesseq.ltrans.N = \x0 \x1 \x2 x2 x1 (\x3 lesseq.rS.N x0 x3)
lesseq.S_is_S.N = \x0 \x1 \x2 lesseq.rec.N (S.total.N x0) x1
  (exists.intro
    (conjunction.intro x0
      (conjunction.intro equal.reflexive (lesseq.refl.N x0))))
  (\x3 \x4 \x5 exists.elim
    (\x6 conjunction.left
      (\x7 \x8 conjunction.left
        (\x9 \x10 exists.intro
          (conjunction.intro (x9 (\x11 x11) (S.total.N x7))
            (conjunction.intro equal.reflexive
              (x9 (\x11 x11) (lesseq.rS.N x0 x10))))) x8) x6) x5)
  x2
lesseq.S_is_S_left.N = \x0 \x1 \x2 \x3 exists.elim
  (\x4 conjunction.left
    (\x5 \x6 conjunction.left (\x7 \x8 x2 x5 x7 x8) x6) x4)
  (lesseq.S_is_S.N x0 x1 x3)
lem2 = \x0 \x1 \x2 \x3
  (\x4 exists.elim
    (\x5 conjunction.left
      (\x6 \x7 conjunction.left
        (\x8 \x9 conjunction.left
          (\x10 \x11 exists.intro
            (conjunction.intro x6
              (conjunction.intro x10
                (conjunction.intro x11
                  (\x12 \x13 \x14 x9 x12
                    (lesseq.S_is_S_left.N x6 x12
                      (\x15 \x16 \x17 conjunction.intro
                        (x16 (\x18 x18)
                          (lesseq.rS.N x3
                            (lesseq.ltrans.N x3 x10 x17))) x13)
                      x14)))))) x8) x7) x5)
    (lem1
      (exists.elim
        (\x5 conjunction.left
          (\x6 \x7 \x8 conjunction.left
            (\x9 \x10 exists.elim (\x11 x0 x10) (x2 x6)) x8) x5)
        x4) x1 x4)) (x2 x3)
cons.total.List = \x0 \x1 \x2 \x3 x3 x0 (x1 x2 x3)
nil.total.List = \x0 \x1 x0
rec.List = \x0 \x1 \x2 conjunction.left (\x3 \x4 x4)
  (x2 (conjunction.intro nil.total.List x0)
    (\x3 \x4 conjunction.left
      (\x5 \x6 conjunction.intro (cons.total.List x3 x5)
        (x1 x3 x5 x6)) x4))
lem3 = \x0 rec.List
  (\x1 \x2 exists.intro
    (conjunction.intro (\x3 x3)
      (conjunction.intro x2 (\x3 \x4 \x5 nil.total.List))))
  (\x1 \x2 \x3 \x4 \x5 exists.elim
    (\x6 conjunction.left
      (\x7 \x8 conjunction.left
        (\x9 \x10
          (\x11 exists.intro
            (conjunction.intro
              (\x12 conjunction.left
                (\x13 \x14 exists.elim (\x15 x7 x13)
                  (x11 N0.total.N)) x12)
              (conjunction.intro x11
                (\x12 \x13 \x14 cons.total.List
                  (conjunction.right.elim x12
                    (conjunction.left (\x15 \x16 x4 (x7 x15)) x13)
                    (conjunction.left (\x15 \x16 x15) x13) x14)
                  (x10 (conjunction.left (\x15 \x16 x15) x12)
                    (conjunction.left (\x15 \x16 x15) x13) x14)))))
          (lem2 (\x11 x4 (x7 x11)) x1 x9)) x8) x6) (
    x3 x4 x5)) x0
List_increasing = \x0 \x1 rec.List nil.total.List
  (\x2 \x3 \x4 cons.total.List (x0 x2) x4) x1
succ.Good = \x0 \x1 \x2 \x3 \x4 \x5 \x6 \x7 x7 x0 x1 x2
  (x3 x5 x6 x7) x4
true.intro = \x0 x0
True.List = \x0 rec.List nil.total.List
  (\x1 \x2 \x3 cons.total.List true.intro x3) x0
un.Good = \x0 \x1 \x2 \x3 x2 x0
case.List = \x0 x0 (disjunction.left.intro equal.reflexive)
  (\x1 \x2 disjunction.right.intro
    (disjunction.elim
      (\x3 exists.intro
        (conjunction.intro x1
          (exists.intro
            (conjunction.intro (x3 (\x4 x4) nil.total.List)
              equal.reflexive))))
      (\x3 exists.elim
        (\x4 conjunction.left
          (\x5 \x6 exists.elim
            (\x7 conjunction.left
              (\x8 \x9 exists.intro
                (conjunction.intro x1
                  (exists.intro
                    (conjunction.intro
                      (x9 (\x10 x10) (cons.total.List x5 x8))
                      equal.reflexive)))) x7) x6) x4) x3) x2))
case_left.List = \x0 \x1 \x2 disjunction.elim
  (\x3 x3 (\x4 x4) (x0 x3))
  (\x3 exists.elim
    (\x4 conjunction.left
      (\x5 \x6 exists.elim
        (\x7 conjunction.left
          (\x8 \x9 x9 (\x10 x10) (x1 x5 x8 x9)) x7) x6) x4) x3)
  (case.List x2)
zero.Good = \x0 \x1 \x2 x0
rec.Good = \x0 \x1 \x2 \x3 conjunction.left (\x4 \x5 x5)
  (x3 (conjunction.intro zero.Good x0)
    (\x4 conjunction.intro (un.Good x4) (x1 x4))
    (\x4 \x5 \x6 \x7 \x8 conjunction.intro
      (succ.Good x4 x5 x6 (conjunction.left (\x9 \x10 x9) x7) x8)
      (x2 x4 x5 x6 (conjunction.left (\x9 \x10 x9) x7)
        (conjunction.left (\x9 \x10 x10) x7) x8)))
dickson = \x0 \x1 exists.elim
  (\x2 exists.elim
    (\x3 conjunction.left
      (\x4 \x5 conjunction.left
        (\x6 \x7 exists.intro
          (conjunction.intro
            (rec.Good nil.total.List
              (\x8 cons.total.List x8 nil.total.List)
              (\x8 \x9 \x10 \x11 \x12 \x13 cons.total.List x8 x12)
              x7) x5)) x5) x3)
    (rec.N
      (exists.intro
        (conjunction.intro nil.total.List
          (conjunction.intro (length.nil.List equal.reflexive)
            zero.Good)))
      (\x3 \x4 exists.elim
        (\x5 conjunction.left
          (\x6 \x7 conjunction.left
            (\x8 \x9 conjunction.left
              (\x10 \x11 case_left.List
                (\x12 exists.elim
                  (\x13 conjunction.left
                    (\x14 \x15 conjunction.left
                      (\x16 \x17 exists.intro
                        (conjunction.left
                          (\x18 \x19 conjunction.intro
                            (cons.total.List x17 nil.total.List)
                            (conjunction.intro
                              (x18
                                (x12 (\x20 x20)
                                  (length.cons.List nil.total.List
                                    equal.reflexive)))
                              (un.Good x14))) x7)) x15) x13)
                  (x10 N0.total.N))
                (\x12 \x13 \x14 exists.elim
                  (\x15 conjunction.left
                    (\x16 \x17 conjunction.left
                      (\x18 \x19 exists.intro
                        (lesseq.S_is_S_left.N (x8 x12) x16
                          (\x20 \x21 \x22 conjunction.left
                            (\x23 \x24 conjunction.intro
                              (cons.total.List x19 x6)
                              (conjunction.intro
                                (x23
                                  (length.cons.List
                                    (x14 (\x25 x25)
                                      (cons.total.List true.intro
                                        (True.List x13)))
                                    equal.reflexive))
                                (x14 (\x25 x25)
                                  (succ.Good x16
                                    (exists.elim (\x25 x8 x12)
                                      (x10 x20))
                                    (List_increasing x8 x13)
                                    (x14 x24)
                                    (conjunction.intro
```

```
                                    (x21 (\x25 x25)
                                      (lesseq.lS.N
                                        (exists.elim
                                          (\x25 x8 x12)
                                          (x10 x20)) x22))
                                    (x11 x12 x19
                                      (x21 (\x25 x25)
                                        (lesseq.lS.N (
                                          x8 x12) x22)))))))) x7)
                            x18)) x17) x15)
                    (x10 (S.total.N (x8 x12)))) x6) x9) x2) x5) x4)
      x1))
  (lem3 x0 (\x2 x2)
    (\x2 exists.intro
      (conjunction.intro x2
        (conjunction.intro (lesseq.refl.N x2) x2))))
NStor = \x0 x0 (\x1 x1 N0.total.N)
  (\x1 \x2 x1 (\x3 x2 (S.total.N x3)))
LNStor = \x0 x0 (\x1 x1 nil.total.List)
  (\x1 \x2 \x3 NStor x1 (\x4 x2 (\x5 x3 (cons.total.List x4 x5))))
conjunction.left.elim = \x0 x0 (\x1 \x2 x1)
cons.injective_left.List = \x0 \x1 x0
  (conjunction.left.elim (cons.injective.List x1))
  (conjunction.right.elim (cons.injective.List x1))
cons_not_nil.List = \x0 nil_not_cons.List
  (x0 (\x1 x1) equal.reflexive)
case.Good = \x0 \x1 \x2 \x3 rec.Good
  (\x4 \x5 \x6 x4 equal.reflexive)
  (\x4 \x5 \x6 \x7 x6 x4 equal.reflexive)
  (\x4 \x5 \x6 \x7 \x8 \x9 \x10 \x11 \x12 x12 x4 x5 x6 x7 x9
    equal.reflexive) x3 x0 x1 x2
length.total.List = \x0 rec.List
  (length.nil.List (\x1 x1) N0.total.N)
  (\x1 \x2 \x3 length.cons.List x2 (\x4 x4) (S.total.N x3)) x0
lesseq.anti_sym.N = \x0 rec.N
  (\x1 \x2 \x3 lesseq.rN0_left.N x1 (\x4 x4 (\x5 x5) (x4 x4)) x3)
  (\x1 \x2 \x3 rec.N
    (\x4 lesseq.rN0_left.N (S.total.N x1) (\x5 S_not_N0.N x1 x5)
      x4)
    (\x4 \x5 \x6 \x7 x2 x4 (lesseq.S_inj_left.N x1 x4 (\x8 x8) x6)
      (lesseq.S_inj_left.N x4 x1 (\x8 x8) x7) equal.reflexive) x3)
  x0
x_not_Sx.N = \x0 rec.N (\x1 N0_not_S.N N0.total.N x1)
  (\x1 \x2 \x3 S_inj_left.N x1 (S.total.N x1) (\x4 x2 x4) x3) x0
lesseq.Sx_x.N = \x0 \x1 x_not_Sx.N x0
  (lesseq.anti_sym.N x0 (S.total.N x0)
    (lesseq.rS.N x0 (lesseq.refl.N x0)) x1)
less_dec = \x0 rec.N
  (\x1 case_left.N
    (\x2 disjunction.right.intro (\x3 lesseq.Sx_x.N (x2 x1) x3))
    (\x2 \x3 disjunction.left.intro
      (lesseq.lS.N N0.total.N (lesseq.lN0.N x2))) x1)
  (\x1 \x2 \x3 rec.N
    (disjunction.right.intro (\x4 lesseq.rN1.N (S.total.N x1) x4))
    (\x4 \x5 disjunction.elim
      (\x6 disjunction.left.intro
        (lesseq.S_is_S_left.N x1 x4
          (\x7 \x8 \x9 lesseq.lS.N (S.total.N x1)
            (x8 (\x10 x10) (lesseq.lS.N x1 x9))) x6))
      (\x6 disjunction.right.intro
        (\x7 lesseq.S_inj_left.N (S.total.N x1) x4
          (\x8 lesseq.S_is_S_left.N x1 x4
            (\x9 \x10 \x11 x6
              (x10 (\x12 x12) (lesseq.lS.N x1 x11))) x8) x7))
      (x2 x4)) x3) x0
lesseq_dec = \x0 rec.N
  (\x1 disjunction.left.intro (lesseq.lN0.N x1))
  (\x1 \x2 \x3 rec.N
    (disjunction.right.intro (\x4 lesseq.rN1.N x1 x4))
    (\x4 \x5 disjunction.elim
      (\x6 disjunction.left.intro (lesseq.lS.N x1 x6))
      (\x6 disjunction.right.intro
        (\x7 lesseq.S_inj_left.N x1 x4 (\x8 x6 x8) x7)) (
      x2 x4)) x3) x0
cons.left.List = \x0 \x1 case_left.List (\x2 cons_not_nil.List x2)
  (\x2 \x3 \x4 cons.injective_left.List
    (\x5 \x6 x0 (x5 (\x7 x7) x2) (x6 (\x7 x7) x3)) x4) x1
forall_dec = \x0 \x1 rec.List
  (disjunction.left.intro nil.total.List)
  (\x2 \x3 \x4 disjunction.elim
    (\x5 disjunction.elim
      (\x6 disjunction.left.intro (cons.total.List x6 x5))
      (\x6 disjunction.right.intro
        (\x7 cons.left.List (\x8 \x9 x6 x8) x7)) (x0 x2))
    (\x5 disjunction.right.intro
      (\x6 cons.left.List (\x7 \x8 x5 x8) x6)) x4) x1
Spec_dec = \x0 \x1 \x2 rec.List
  (\x3 case_left.N
    (\x4 disjunction.left.intro
      (conjunction.intro (length.nil.List equal.reflexive)
        zero.Good))
    (\x4 \x5 disjunction.right.intro
      (\x6 conjunction.left
        (\x7 \x8 N0_not_S.N x4 (length.nil.List x7)) x6)) x3)
  (\x3 \x4 \x5 \x6 case_left.N
    (\x7 disjunction.right.intro
      (\x8 conjunction.left
        (\x9 \x10 S_not_N0.N (length.total.List (True.List x4))
          (length.cons.List (True.List x4) x9)) x8))
    (\x7 \x8 disjunction.elim
      (\x9 case_left.List
        (\x10 disjunction.left.intro
          (conjunction.left
            (\x11 \x12 conjunction.intro
              (x11
                (x10 (\x13 x13)
                  (length.cons.List nil.total.List equal.reflexive)))
              (un.Good x3)) x9))
        (\x10 \x11 \x12 disjunction.elim
          (\x13 disjunction.elim
            (\x14 disjunction.left.intro
              (lesseq.S_is_S_left.N x10 x3
                (\x15 \x16 \x17 conjunction.left
                  (\x18 \x19 conjunction.intro
                    (x18
                      (x12 (\x20 x20)
                        (length.cons.List
                          (cons.total.List true.intro
                            (True.List x11)) equal.reflexive)))
                    (succ.Good x3 x10 x11 (x12 x19)
                      (conjunction.intro
                        (x16 (\x20 x20) (lesseq.lS.N x10 x17)) x14)))
                  x9) x13))
            (\x14 disjunction.right.intro
              (\x15 conjunction.left
                (\x16 \x17 case.Good (\x18 cons_not_nil.List x18)
                  (\x18 \x19 cons.injective_left.List
                    (\x20 \x21 cons_not_nil.List x21) x19)
                  (\x18 \x19 \x20 \x21 \x22 \x23
                    cons.injective_left.List
                    (\x24 \x25 cons.injective_left.List
                      (\x26 \x27 conjunction.left
                        (\x28 \x29 x14
                          (x24 (\x30 x30) (x26 (\x30 x30) x29)))
                        x22) x25) x23) x17) x15))
              (forall_dec (\x14 lesseq_dec (x14 x10) (x14 x3)) x0))
            (\x13 disjunction.right.intro
              (\x14 conjunction.left
                (\x15 \x16 case.Good (\x17 cons_not_nil.List x17)
                  (\x17 \x18 cons.injective_left.List
                    (\x19 \x20 cons_not_nil.List x20) x18)
                  (\x17 \x18 \x19 \x20 \x21 \x22
                    cons.injective_left.List
                    (\x23 \x24 cons.injective_left.List
                      (\x25 \x26 conjunction.left
                        (\x27 \x28 lesseq.S_is_S_left.N x18 x17
                          (\x29 \x30 \x31 x13
                            (x23 (\x32 x32)
                              (x30 (\x32 x32)
                                (lesseq.lS.N x10
                                  (x25 (\x32 x32) x31))))) x27) x21)
                        x24) x22) x16) x14)) (less_dec x10 x3)) x4)
        (\x9 disjunction.right.intro
          (\x10 x9
            (conjunction.left
              (\x11 \x12 conjunction.intro
                (S_inj_left.N (length.total.List (True.List x4)) x7
                  (\x13 x13) (length.cons.List (True.List x4) x11))
                (case.Good (\x13 cons_not_nil.List x13)
                  (\x13 \x14 cons.injective_left.List
                    (\x15 \x16 x16 (\x17 x17) zero.Good) x14)
                  (\x13 \x14 \x15 \x16 \x17 \x18
                    cons.injective_left.List
                    (\x19 \x20 x20 (\x21 x21) x16) x18) x12)) x10)))
        (x5 x7)) x6) x2 x1
PN = \x0 x0 "0" "S"
PL = \x0 x0 "\n" (\x1 \x2 PN x1 "," x2)
interact = \x0 \x1 dickson x0 x1
  (\x2 x2
    (\x3 \x4 LNStor x3
      (\x5 PL x5 (Spec_dec x0 x1 x5 (\x6 x5) (\x6 x6 x4)))))
```