Program development by proof transformation

Ulrich Berger and Helmut Schwichtenberg

Mathematisches Institut der Universität München

We begin by reviewing the natural deduction rules for the $\to\wedge\forall$–fragment of minimal logic. It is shown how intuitionistic and classical logic can be embedded. Recursion and induction is added to obtain a more realistic proof system. Simple types are added in order to make the language more expressive. We also consider two alternative methods to deal with the strong or constructive existential quantifier $\exists^*$. Finally we discuss the well–known notion of an extracted program of a derivation involving $\exists^*$, in order to set up a relation between the two alternatives.

Section 2 deals with the computational content of classical proofs. As is well–known a proof of a $\forall\exists$–theorem with a quantifier–free kernel — where $\exists$ is viewed as defined by $\neg\forall\neg$ — can be used as a program. We describe a "direct method" to use such a proof as a program, and compare it with Harvey Friedman's A–translation [3] followed by program extraction from the resulting constructive proof. It is shown that both algorithms coincide.

In section 3 Goad's method of pruning of proof trees is introduced. It is shown how a proof can be simplified after addition of some further assumptions. In a first step some subproofs are replaced by different ones using the additional assumptions. In a second step parts of the proof tree are pruned, i.e. cut out. Note that the first step involves searching for new proofs — using the new assumptions — of formulas in the proof tree. Hence we also have to discuss proof search in minimal logic.

Finally section 4 treats an example already considered by Goad in his thesis [5], the binpacking problem. The main difference to Goad's work is that he used a logic with the strong existential quantifier, whereas we work within the $\to\forall$–fragment. This example is particularly well–suited to demonstrate that the pruning method can be applied to adapt programs to particular situations, and moreover that pruning can change the functions computed by programs. In this sense this method is essentially different from program development by program transformation.

We would like to thank Michael Bopp and Karl–Heinz Niggl for their help in preparing these notes.

1. Proofs in minimal logic and arithmetic

Let us first fix our language $\mathcal{L}$. Let G be a set of ground types (e.g. nat and boole). *Types* (also called object types or simple types) are formed from G by the operations $\rho \to \sigma$ and $\rho \times \sigma$. For any type ρ let a countable infinite set of *variables* of type ρ be given. We denote variables of type ρ by $x^\rho, y^\rho, \dots$. We also assume that a set $\mathcal{C}$ of *constants* denoted c^ρ is given, each of an arbitrary type ρ.

Terms and their types are defined inductively by

- x^ρ and c^ρ are terms of type ρ,
- If r is a term of type $\rho \to \sigma$ and s is a term of type ρ, the rs is a term of type σ.
- If r is a term of type σ, then $\lambda x^\rho\, r$ is a term of type $\rho \to \sigma$.
- If t_i is a term of type ρ_i for $i \in \{0, 1\}$, then $\langle t_0, t_1 \rangle$ is a term of type $\rho_0 \times \rho_1$.
- If t is a term of type $\rho_0 \times \rho_1$ and $i \in \{0, 1\}$, then $\pi_i(t)$ is a term of type ρ_i.

The set $\mathrm{FV}(r)$ of *free variables* of a term r is defined as usual. It is well known that any term has a unique normal form with respect to $\beta\eta$–conversion. In the sequel we will identify terms with the same $\beta\eta$-normal form.

We now fix our notion of a formula. We assume that a set $\mathcal{P}$ of *predicate symbols* P of arities $\rho_1, \dots, \rho_m$ is given. 0–ary relation symbols are called propositional symbols. *Formulas* are defined inductively by

- If $t_1^{\rho_1}, \dots, t_n^{\rho_n}$ are terms and $R \in \mathcal{P}$ is a relation symbol of arity $\rho_1, \dots, \rho_n$, then $R(t_1, \dots, t_n)$ is a formula.
- $\bot$ (to be read "falsity") is a formula.
- If A and B are formulas, then $A \to B$ is a formula.
- If A and B are formulas, then $A \land B$ is a formula.
- If A is a formula and x^ρ is a variable, then $\forall x^\rho\, A$ is a formula.

$R(t_1, \dots, t_n)$ and $\bot$ are called *atomic formulas* or *atoms*. Note that $\bot$ is not a propositional symbol.

We use (also with indices)

$$\begin{aligned}
r, s, t &\quad \text{for terms,}\\
x, y, z &\quad \text{for variables,}\\
a, b, c &\quad \text{for constants,}\\
P, Q, R &\quad \text{for relation symbols,}\\
f, g, h &\quad \text{for function symbols,}\\
A, B, C &\quad \text{for formulas.}
\end{aligned}$$

Negation, disjunction and the existential quantifier are defined by

$$\begin{aligned}
\neg A &:= A \to \bot,\\
A \lor B &:= \neg A \land \neg B \to \bot,\\
\exists x\, A &:= \neg \forall x\, \neg A.
\end{aligned}$$

A term t is called *closed*, if $\mathrm{FV}(t) = \emptyset$. We write $t[x_1, \dots, x_n]$ to indicate that all variables free in t are in the list $x_1, \dots, x_n$. Similar conventions apply to formulas.

As our deductive formalism we use the system of natural deduction introduced by Gerhard Gentzen in [4]. It consists of the following introduction and elimination rules for $\to$, $\wedge$ and $\forall$.

For any $\mathcal{L}$–formula A let countably many *assumption variables* of type A be given. We use u^A, v^A, w^A to denote assumption variables of type A.

The notions of a *derivation term* d^A in minimal logic and its set $\mathrm{FA}(d^A)$ of *free assumption variables* are defined inductively by

(A) u^A is a derivation term with $\mathrm{FA}(u^A) = \{u^A\}$.

($\to^+$) If d^B is a derivation term, then

$$(\lambda u^A\, d^B)^{A \to B}$$

is a derivation term with $\mathrm{FA}(\lambda u^A\, d^B) = \mathrm{FA}(d^B) \setminus \{u^A\}$.

($\to^-$) If $d^{A \to B}$ and e^A are derivation terms, then

$$(d^{A \to B} e^A)^B$$

is a derivation term with $\mathrm{FA}(d^{A \to B} e^A) = \mathrm{FA}(d^{A \to B}) \cup \mathrm{FA}(e^A)$.

($\wedge^+$) If d^B and e^B are derivation terms, then

$$\langle d^A, e^B \rangle^{A \wedge B}$$

is a derivation term with $\mathrm{FA}(\langle d^A, e^B \rangle^{A \wedge B}) := \mathrm{FA}(d^A) \cup \mathrm{FA}(e^B)$.

($\wedge^-$) If $d^{A_0 \wedge A_1}$ is a derivation term, then

$$\pi_i(d^{A_0 \wedge A_1})^{A_i}$$

is a derivation term with $\mathrm{FA}(\pi_i(d^{A_0 \wedge A_1})^{A_i}) := \mathrm{FA}(d^{A_0 \wedge A_1})$ (for $i \in \{0, 1\}$).

($\forall^+$) If d^A is a derivation term and $x^\rho \notin \bigcup\{\mathrm{FV}(B) | u^B \in \mathrm{FA}(d^A)\}$, then

$$(\lambda x^\rho\, d^A)^{\forall x^\rho\, A}$$

is a derivation term with $\mathrm{FA}(\lambda x^\rho\, d^A) = \mathrm{FA}(d^A)$.

($\forall^-$) If $d^{\forall x^\rho\, A}$ is a derivation term and t^ρ is a term, then

$$(d^{\forall x^\rho\, A} t^\rho)^{A[t/x^\rho]}$$

is a derivation term with $\mathrm{FA}(d^{\forall x^\rho\, A} t^\rho) = \mathrm{FA}(d^{\forall x^\rho\, A})$.

Recall that we identify terms with the same $\beta\eta$-normal form. Hence e.g. $d^{R(r) \to B} e^{R(s)}$ is a correct derivation provided r and s have the same $\beta\eta$-normal forms.

A derivation term d^A is called *closed*, if $\mathrm{FA}(d^A) = \emptyset$. We write

$$d^B[u_1^{A_1}, \dots, u_n^{A_n}]$$

to indicate that the assumption variables free in d^B are in the list $u_1^{A_1}, \dots, u_n^{A_n}$. We also use the notation $d\colon A$ instead of d^A.

Definition. *A formula A is called derivable from assumptions $A_1, \dots, A_n$, is there is a derivation term $d^B[u_1^{A_1}, \dots, u_n^{A_n}]$ with different assumption variables $u_1^{A_1}, \dots, u_n^{A_n}$.*

Let S be a (finite or infinite) set of formulas. We write $S \vdash B$, if the formula B is derivable from finitely many assumptions $A_1, \dots, A_n \in S$.

Examples for derivable formulas are

$$\begin{aligned}
&P \to (Q \to P),\\
&(P \to Q \to R) \to (P \to Q) \to P \to R,\\
&(\forall x. P(x) \to Q(x)) \to \forall x\, P(x) \to \forall x\, Q(x).
\end{aligned}$$

Definition. *For any derivation d we define its set $\mathrm{FV}(d)$ of free (object) variables by*

$$\begin{aligned}
\mathrm{FV}(u^A) &:= \mathrm{FV}(A).\\
\mathrm{FV}(\lambda u^A d^B) &:= \mathrm{FV}(A) \cup \mathrm{FV}(d^B).\\
\mathrm{FV}(d^{A\to B} e^A) &:= \mathrm{FV}(d^{A\to B}) \cup \mathrm{FV}(e^A).\\
\mathrm{FV}(\langle d^A, e^B \rangle) &:= \mathrm{FV}(d^A) \cup \mathrm{FV}(e^B).\\
\mathrm{FV}(\pi_i(d^{A\wedge B})) &:= \mathrm{FV}(d^{A\wedge B}).\\
\mathrm{FV}(\lambda x\, d^A) &= \mathrm{FV}(d^A) \setminus \{x\}.\\
\mathrm{FV}(d^{\forall x\, A} t) &:= \mathrm{FV}(d^{\forall x\, A}) \cup \mathrm{FV}(t).
\end{aligned}$$

Example: Let d be the derivation given by

$$\frac{u\colon R(x)}{R(x) \to\ R(x)} \to^+ u$$

Then we have

$$\begin{aligned}
d &= \lambda u^{R(x)}\, u^{R(x)},\\
\mathrm{FA}(d) &= \emptyset,\\
\mathrm{FV}(d) &= \{x\}.
\end{aligned}$$

For derivation terms d we have two kinds of substitution: we can substitute a derivation term f^A for a free assumption variable u^A, denoted $d[f/u]$, and we can substitute an object term t for a free object variable x, denoted $d[t/x]$.

Lemma. *(i) If $d\colon A$, $f\colon B$ are derivation terms and t is an object term, then $d[f/u^B]\colon A$ and $d[t/x]\colon A[t/x]$ are derivation terms.*

(ii) $\mathrm{FA}(d[f/u^B]) \subseteq (\mathrm{FA}(d) \setminus \{u^B\}) \cup \mathrm{FA}(f)$.

(iii) $\mathrm{FV}(d[f/u]) \subseteq \mathrm{FV}(d) \cup \mathrm{FV}(f)$.

(iv) $\mathrm{FA}(d[t/x]) = \{v^{A[t/x]} : v^A \in \mathrm{FA}(d)\}$.

(v) $\mathrm{FV}(d[t/x]) \subseteq (\mathrm{FV}(d) \setminus \{x\}) \cup \mathrm{FV}(t)$.

Proof by simultaneous induction (cf. [19], p. 30/31). $\square$

According to this lemma β and η–reduction for derivations are well defined and preserve the derived formula and the free assumptions. Every derivation has a unique $\beta\eta$–normal form.

Derivation terms in *intuitionistic* and in *classical* logic are obtained by adding to the first (assumption–) clause of the definition

- in the case of intuitionistic logic: For any relation symbol $R \in \mathcal{P}$ we let

$$\mathrm{Efq}_R : \forall \vec{x}. \bot \to R(\vec{x})$$

  be a derivation term with $\mathrm{FA}(\mathrm{Efq}_R) = \emptyset$ (Ex–falso–quodlibet axiom).

- in the case of classical logic: For any relation symbol $R \in \mathcal{P}$ we let

$$\mathrm{Stab}_R : \forall \vec{x}. \neg\neg R(\vec{x}) \to R(\vec{x})$$

  be a derivation term with $\mathrm{FA}(\mathrm{Stab}_R) = \emptyset$ (Stability axiom).

Clearly $\mathrm{FV}(\mathrm{Stab}_R) := \mathrm{FV}(\mathrm{Efq}_R) := \emptyset$.

We write $S \vdash A$ ($S \vdash_I A$, $S \vdash_C A$), if there is a derivation term d^A in minimal (intuitionistic, classical) logic such that for any $u^B \in \mathrm{FA}(d)$ we have $B \in S$.

Here are some more interesting examples: The Peirce formula

$$((P \to Q) \to P) \to P$$

is derivable in classical, but not in minimal logic. The Mints formula

$$((((P \to Q) \to P) \to P) \to Q) \to Q$$

is derivable in minimal logic. However, its variant

$$((((P \to Q) \to P) \to P) \to R) \to R$$

is derivable in classical but not in minimal logic.

By obvious reasons the stability axiom is also called the *principle of indirect proof* for the relation symbol R. From our stability axioms we can derive the principle of indirect proof for arbitrary formulas (in our $\to \wedge\forall$–language).

Stability Lemma. *From stability assumptions* Stab_R *for any relation symbol* R *occurring in a formula* A *we can derive* $\neg\neg A \to A$.

Proof by induction on A. $\square$

Similarly from our ex–falso–quodlibet axioms we can derive ex–falso–quodlibet for arbitrary formulas (again in our $\to \wedge\forall$–language).

Ex–falso–quodlibet Lemma. *From assumptions* Efq_R *for any relation symbol* R *occurring in a formula* A *we can derive* $\bot \to A$ *in intuitionistic logic.*

Proof by induction on A. □

From $\neg\neg A \to A$ one can clearly derive $\bot \to A$. Hence any formula derivable in intuitionistic logic is also derivable in classical logic.

We now extend our language $\mathcal{L}$ by a strong existential quantifier written $\exists^*$ (as opposed to $\exists$ defined by $\neg\forall\neg$). There are two approaches to deal with formulas containing $\exists^*$ in a constructive setting, e.g. in minimal or intuitionistic logic.

- (Weyl [20]) A formula containing $\exists^*$ is considered not to be an entity the deduction system can deal with: some "realizing terms" are required to turn it into a "judgement". E.g. r realizes $\exists^* x\, A$ is a judgement, which can be translated into $A[r/x]$.
- (Heyting) The logic is extended by axioms expressing the intended meaning of the strong existential quantifier.

We will treat both approaches here. At first sight, Weyl's point of view is more convincing. However, Heyting's is more prominent in the literature, and we also need it to properly discuss Friedman's A–translation.

Let us first describe Heyting's approach. Here we extend our notion of an $\mathcal{L}$–formula by adding a clause

- If A is a formula, then $\exists^* x^\rho\, A$ is a formula.

In the inductive definition of derivation terms d^A in minimal logic and their sets $\mathrm{FA}(d^A)$ of free assumptions we have to add two more clauses:

($\exists^{*+}$) If A is a formula and x is a variable of type ρ, then

$$\exists^{*+}_{x,A} \colon \forall\forall x.A \to \exists^* x\, A$$

is a derivation term, where $\forall C$ denotes the universal closure of C, and $\mathrm{FA}(\exists^{*+}_{x,A}) = \emptyset$.

($\exists^{*-}$) If A, B are formulas and x is a variable of type ρ such that $x \notin \mathrm{FV}(B)$, then

$$\exists^{*-}_{x,A,B} \colon \forall.(\exists^* x^\rho\, A) \to (\forall x^\rho.A \to B) \to B$$

is a derivation term with $\mathrm{FA}(\exists^{*-}_{x,A,B}) = \emptyset$.

Clearly $\mathrm{FV}(\exists^{*+}) := \mathrm{FV}(\exists^{*-}) := \emptyset$.

For these new derivation terms we have the following conversion rule:

$$\exists^{*-}\vec{t}\left(\exists^{*+}\vec{t}t\, d^{A[t/x]}\right) e \mapsto etd. \qquad (\exists^*)$$

It can be shown that any derivation term has a unique normal form with respect to $\beta\eta\exists^*$–conversion.

Similarly one could also introduce a *strong disjunction* into our natural deduction calculus, written $\vee^*$ (as opposed to $\vee$ defined by $A \vee B := \neg(\neg A \wedge \neg B)$).

Note that one can easily extend the Ex–falso–quodlibet Lemma to the present situation and prove $\bot \to A$ for an arbitrary formula A. In the cases $\exists^* x\, A$ and $A \vee^* B$ just use the corresponding introduction axiom.

Let us now describe Weyl's approach. We restrict ourselves to formulas without $\vee^*$, since later in an arithmetical system we will define $A \vee^* B$ by

$$A \vee^* B := \exists^* p.(p = \mathrm{true} \to A) \wedge (p = \mathrm{false} \to B),$$

where p is a variable of type boole. To every formula A and terms $\vec{r} = r_1^{\rho_1}, \dots, r_m^{\rho_m}$ we associate a *judgement*

$$\vec{r} \,\mathrm{mr}\, A$$

(to be read $\vec{r}$ modified realizes A), which will be a formula not containing $\exists^*$. The list of types $\rho_1, \dots, \rho_m = \tau(A)$ is defined as follows.

$$\tau(R(\vec{t})) := \varepsilon,$$

where ε denotes the empty list, and if $\tau(A) = \vec{\rho}$ and $\tau(B) = \sigma_1, \dots, \sigma_n$ we let

$$\begin{aligned}
\tau(A \to B) &:= \vec{\rho} \to \sigma_1, \dots, \vec{\rho} \to \sigma_n,\\
\tau(A \wedge B) &:= \vec{\rho}, \vec{\sigma},\\
\tau(\forall x^\rho\, B) &:= \rho \to \sigma_1, \dots, \rho \to \sigma_n,\\
\tau(\exists^* x^\rho\, B) &:= \rho, \vec{\sigma}.
\end{aligned}$$

Instead of $\vec{\rho} \to \sigma_1, \dots, \vec{\rho} \to \sigma_n$ we will sometimes write $\vec{\rho} \to \vec{\sigma}$. To give some examples, let n, m, k be of type nat. Then

$$\begin{aligned}
\tau(\forall n \exists^* m\, R(n,m)) &= \mathrm{nat} \to \mathrm{nat},\\
\tau(\forall n \exists^* m \exists^* k\, R(n,m,k)) &= (\mathrm{nat} \to \mathrm{nat}), (\mathrm{nat} \to \mathrm{nat}),\\
\tau(\forall n \exists^* m\, R(n,m) \to \exists^* k\, Q(k)) &= (\mathrm{nat} \to \mathrm{nat}) \to \mathrm{nat}.
\end{aligned}$$

Note that $\tau(A) = \varepsilon$ iff A is a *Harrop formula* (i.e. contains $\exists^*$ in premises of $\to$ only).

Now judgements $\vec{r}^{\tau(A)} \,\mathrm{mr}\, A$ are defined by

$$\begin{aligned}
\varepsilon \,\mathrm{mr}\, R(\vec{t}) &:= R(\vec{t}),\\
r_1, \dots, r_n \,\mathrm{mr}\, (A \to B) &:= \forall \vec{x}. \vec{x} \,\mathrm{mr}\, A \to r_1 \vec{x}, \dots, r_n \vec{x} \,\mathrm{mr}\, B,\\
\vec{r}, \vec{s} \,\mathrm{mr}\, (A \wedge B) &:= \vec{r} \,\mathrm{mr}\, A \wedge \vec{s} \,\mathrm{mr}\, B,\\
r_1, \dots, r_n \,\mathrm{mr}\, \forall x^\rho\, B &:= \forall x^\rho\, r_1 x, \dots, r_n x \,\mathrm{mr}\, B,\\
r, \vec{s} \,\mathrm{mr}\, \exists^* x^\rho\, B &:= \vec{s} \,\mathrm{mr}\, B[r/x].
\end{aligned}$$

Note that for Harrop formulas A we have $\varepsilon \,\mathrm{mr}\, A \equiv A$ iff A does not contain $\exists^*$.

Let us now set up a relation between the implicit and the explicit approach to deal with the existential quantifier.

Definition. *Assume that to any assumption variable u^B we have assigned a list $\vec{x}_u^{\tau(B)} = x_{u1}^{\rho_1}, \dots, x_{un}^{\rho_n}$ of distinct variables, where $\rho_1, \dots, \rho_n = \tau(B)$. Relative to this assignment we define for any derivation d^A its extracted terms $\mathrm{ets}(d^A)$, by induction on d^A. If $\tau(A) = \sigma_1, \dots, \sigma_k$, then $\mathrm{ets}(d^A)$ will be a list $r_1^{\sigma_1}, \dots, r_k^{\sigma_k}$.*

$$
\begin{aligned}
\mathrm{ets}(u^A) &= \vec{x}_u^{\tau(A)}, \\
\mathrm{ets}(\lambda u^A\, d^B) &= \lambda \vec{x}_u^{\tau(A)}\, \mathrm{ets}(d), \\
\mathrm{ets}(d^{A\to B} e^A) &= \mathrm{ets}(d)\mathrm{ets}(e), \\
\mathrm{ets}(\langle d^A, e^B\rangle) &= \mathrm{ets}(d^A), \mathrm{ets}(e^B), \\
\mathrm{ets}(\pi_0(d^{A\wedge B})) &= \textit{the head of } \mathrm{ets}(d^{A\wedge B}) \textit{ of same length as } \tau(A), \\
\mathrm{ets}(\pi_1(d^{A\wedge B})) &= \textit{the tail of } \mathrm{ets}(d^{A\wedge B}) \textit{ of same length as } \tau(B), \\
\mathrm{ets}(\lambda x^\rho\, d^A) &= \lambda x^\rho\, \mathrm{ets}(d), \\
\mathrm{ets}(d^{\forall x^\rho A} t^\rho) &= \mathrm{ets}(d)t, \\
\mathrm{ets}(\exists^{*+}_{x,A}) &= \lambda \vec{x} \lambda x \lambda \vec{y}. x, \vec{y}, \\
\mathrm{ets}(\exists^{*-}_{x,A,B}) &= \lambda \vec{x} \lambda x \lambda \vec{y} \lambda z_1 \dots \lambda z_n . z_1 x \vec{y}, \dots, z_n x \vec{y}.
\end{aligned}
$$

Note that if $\mathrm{ets}(d) = r_1, \dots, r_k$ and $\mathrm{ets}(e) = \vec{s}$, then $\mathrm{ets}(d)\mathrm{ets}(e) = r_1\vec{s}, \dots, r_k\vec{s}$ and $\lambda\vec{x}\,\mathrm{ets}(d) = \lambda\vec{x}\, r_1, \dots, \lambda\vec{x}\, r_k$. In the last clause the (omitted) types are

$$x^\rho, \quad (\vec{y})^{\tau(A)} \quad \text{and} \quad z_j^{\rho\to\tau(A)\to\sigma_j},$$

where $\tau(B) = \sigma_1, \dots, \sigma_n$.

The following can be proved easily.

Lemma. $\mathrm{FV}(\mathrm{ets}(d)) \subseteq \mathrm{FV}(d) \cup \{\vec{x}_u^{\tau(A)} : u^A \in \mathrm{FA}(d)\}$.

Lemma. *We have* $\mathrm{ets}(d[t/x]) = \mathrm{ets}(d)[t/x]$ *and* $\mathrm{ets}(d[e/u]) = \mathrm{ets}(d)[\mathrm{ets}(e)/\vec{x}_u]$, *hence*

$$
\begin{aligned}
d =_{\beta\exists^*} e \quad &\Longrightarrow \quad \mathrm{ets}(d) =_\beta \mathrm{ets}(e) \\
d =_\eta e \quad &\Longrightarrow \quad \mathrm{ets}(d) =_\eta \mathrm{ets}(e)
\end{aligned}
$$
□

Hence we can safely identify derivations with the same $\beta\eta\exists^*$–normal forms.

Soundness Theorem. *Assume that to any assumption variable u^B we have assigned a list $\vec{x}_u^{\tau(B)}$ and a new assumption variable $\tilde{u}\colon \vec{x}_u^{\tau(B)}\, \mathrm{mr}\, B$. Relative to this assignment we can find for any derivation $d\colon A$ a derivation*

$$\mu(d)\colon \mathrm{ets}(d)\, \mathrm{mr}\, A$$

with $\mathrm{FA}(\mu(d)) \subseteq \{\tilde{u}\colon \vec{x}_u\, \mathrm{mr}\, B \mid u\colon B \in \mathrm{FA}(d)\}$.

Proof by induction on d.

Case $u_i\colon A_i$. Then $\tilde{u}_i\colon \vec{x}_{u_i}\, \mathrm{mr}\, A_i$.

Case $\lambda u^B\, d^A$. Let $\tau(B) = \vec{\rho}$ and $\tau(A) = \sigma_1, \dots, \sigma_m$. Then we have $\tau(B \to A) = \vec{\rho} \to \sigma_1, \dots, \vec{\rho} \to \sigma_m$. We look for a derivation of

$$\lambda \vec{x}^{\vec{\rho}}\, \mathrm{ets}(d^A)\, \mathrm{mr}\, (B \to A),$$

i.e. of

$$\forall \vec{x}^{\vec{\rho}}. \vec{x}\, \mathrm{mr}\, B \to \mathrm{ets}(d^A)\, \mathrm{mr}\, A,$$

since $(\lambda \vec{x}^{\vec{\rho}}\, \mathrm{ets}(d^A))\vec{x} =_\beta \mathrm{ets}(d^A)$ and terms with the same β–normal form are identified. Hence we can take

$$\mu(\lambda u^B\, d^A) := \lambda \vec{x} \lambda \tilde{u}^{\vec{x}\, \mathrm{mr}\, B} \mu(d).$$

Case $d^{A \to B} e^A$. By IH we have $\mu(d)\colon \mathrm{ets}(d)\, \mathrm{mr}\, (A \to B)$, $\mu(e)\colon \mathrm{ets}(e)\, \mathrm{mr}\, A$. By definition the first of these means

$$\mu(d)\colon \forall \vec{x}. \vec{x}\, \mathrm{mr}\, A \to \mathrm{ets}(d)\vec{x}\, \mathrm{mr}\, B.$$

Hence we have $\mu(d)\mathrm{ets}(e)\mu(e)\colon \mathrm{ets}(d)\mathrm{ets}(e)\, \mathrm{mr}\, B$, and since $\mathrm{ets}(de) = \mathrm{ets}(d)\mathrm{ets}(e)$ we can take $\mu(de) := \mu(d)\mathrm{ets}(e)\mu(e)$.

Cases $\langle d^A, e^B \rangle$, $\pi_i(d^{A \wedge B})$. Easy.

Case $\lambda x^\rho\, d^A$. We have to find a derivation of $\lambda x^\rho \mathrm{ets}(d)\, \mathrm{mr}\, \forall x^\rho A$, i.e. of (since we identify terms with the same β–normal forms) $\forall x^\rho\, \mathrm{ets}(d)\, \mathrm{mr}\, A$. By IH we have $\mu(d)\colon \mathrm{ets}(d)\, \mathrm{mr}\, A$ with free assumptions not involving x. Hence we can take $\mu(\lambda x^\rho\, d) := \lambda x^\rho\, \mu(d)$.

Case $d^{\forall x^\rho A} t^\rho$. By IH we have

$$\mu(d)\colon \mathrm{ets}(d)\, \mathrm{mr}\, \forall x\, A,$$

i.e.

$$\mu(d)\colon \forall x\, \mathrm{ets}(d) x\, \mathrm{mr}\, A.$$

Hence $\mu(d)t\colon \mathrm{ets}(d)t\, \mathrm{mr}\, A[t/x]$ and we can take $\mu(dt) := \mu(d)t$.

Case $\exists^{*+}_{x,A}$. We look for a derivation $\mu(\exists^{*+}_{x,A})$ of

$$\begin{aligned}
&(\lambda \vec{x} \lambda x \lambda \vec{y}. x, \vec{y})\, \mathrm{mr}\, \forall \vec{x} \forall x^\rho. A \to \exists^* x^\rho\, A \\
&\forall \vec{x} \forall x^\rho (\lambda \vec{y}. x, \vec{y})\, \mathrm{mr}\, (A \to \exists^* x^\rho\, A) \\
&\forall \vec{x} \forall x^\rho \forall \vec{y}. \vec{y}\, \mathrm{mr}\, A \to x, \vec{y}\, \mathrm{mr}\, \exists^* x^\rho\, A \\
&\forall \vec{x} \forall x^\rho \forall \vec{y}. \vec{y}\, \mathrm{mr}\, A \to \vec{y}\, \mathrm{mr}\, A
\end{aligned}$$

Hence we can take $\mu(\exists^{*+}_{x,A}) := \lambda \vec{x} \lambda x \lambda \vec{y} \lambda u\, u$ where $u\colon \vec{y}\, \mathrm{mr}\, A$.

Case $\exists^{*-}_{x,A,B}$. Recall that

$$\exists^{*-}_{x,A,B}\colon \forall \vec{x}. \exists^* x\, A \to (\forall x. A \to B) \to B$$

with $x \notin \mathrm{FV}(B)$. We look for a derivation $\mu(\exists^{*-}_{x,A,B})$ of

$$\begin{aligned}
&(\lambda \vec{x} \lambda x \lambda \vec{y} \lambda z_1 \dots \lambda z_n. z_1 x \vec{y}, \dots, z_n x \vec{y})\, \mathrm{mr}\, \forall \vec{x}. \exists^* x\, A \to (\forall x. A \to B) \to B \\
&\forall \vec{x} (\lambda x \lambda \vec{y} \lambda z_1 \dots \lambda z_n. z_1 x \vec{y}, \dots, z_n x \vec{y})\, \mathrm{mr}\, (\exists^* x\, A \to (\forall x. A \to B) \to B) \\
&\forall \vec{x} \forall x \forall \vec{y}. x, \vec{y}\, \mathrm{mr}\, \exists^* x\, A \to (\lambda z_1 \dots \lambda z_n. z_1 x \vec{y}, \dots, z_n x \vec{y})\, \mathrm{mr}\, ((\forall x. A \to B) \to B) \\
&\forall \vec{x} \forall x \forall \vec{y}. \vec{y}\, \mathrm{mr}\, A \to \forall \vec{z}. \vec{z}\, \mathrm{mr}\, (\forall x. A \to B) \to z_1 x \vec{y}, \dots, z_n x \vec{y}\, \mathrm{mr}\, B
\end{aligned}$$

Now since $\vec{z}\,\mathrm{mr}\,\forall x.A \to B$ is the same as

$$\forall x \forall \vec{y}.\vec{y}\,\mathrm{mr}\,A \to z_1 x\vec{y}, \dots, z_n x\vec{y}\,\mathrm{mr}\,B$$

we can easily derive this formula by

$$\mu(\exists^{*-}_{x,A,B}) := \lambda\vec{x}\lambda x\lambda\vec{y}\lambda u\lambda\vec{z}\lambda v.vx\vec{y}u,$$

where $u\colon \vec{y}\,\mathrm{mr}\,A$ and $v\colon \forall x\forall\vec{y}.\vec{y}\,\mathrm{mr}\,A \to z_1 x\vec{y}, \dots, z_n x\vec{y}\,\mathrm{mr}\,B$. □

Remark: Mints has shown already in [8] that for any $d\colon \exists^* x\, A$ with $\mathrm{FA}(d) = \mathrm{FV}(d) = \emptyset$ the extracted terms $\mathrm{ets}(d)$ reduce into the terms that can be read off from the long normal form of $d\colon \exists^* x\, A$. This has later been generalized by Stein in his thesis [16].

Let us now extend these considerations to arithmetic. We allow constants for primitive recursive functionals of arbitrary types (i.e. terms of Gödel's T), identifying terms with the same normal form (w.r.t. the usual conversion rules for Gödel's T). It is assumed that at least the ground types nat of natural numbers and boole of booleans are present. We restrict ourselves to decidable atomic formulas; it is convenient to represent them by boolean terms, i.e. in the form $\mathrm{atom}(t^{\mathrm{boole}})$ where atom is a distinguished relation symbol. We could equally well take equations $r = s$ with r, s terms of type nat as the only boolean formulas. We do not need $\bot$ as an extra atomic formula, since it can be defined by $\bot := \mathrm{atom}(\mathrm{false})$. Let us use n, m as variables of type nat and p, q as variables of type boole. Our *induction schemata* are the universal closures of

$$A[0/n] \to (\forall n.A \to A[n+1/n]) \to \forall n\, A,$$
$$A[\mathrm{true}/p] \to A[\mathrm{false}/p] \to \forall p\, A.$$

We can use boolean induction (i.e. case analysis) to prove stability $\neg\neg\mathrm{atom}(p) \to \mathrm{atom}(p)$ and from this we can as before conclude the stability $\neg\neg A \to A$ of formulas A built from atoms by $\to$, $\wedge$ and $\forall$. As already remarked, strong disjunction $\vee^*$ can be defined by means of the strong existential quantifier $\exists^*$.

Let us now carry out this program. First we extend the notion of a term by adding the clauses

- The constants 0^{nat}, $S^{\mathrm{nat}\to\mathrm{nat}}$, $\mathrm{true}^{\mathrm{boole}}$ and $\mathrm{false}^{\mathrm{boole}}$ are terms.
- For each type ρ the constants

  $$R_{\mathrm{nat},\rho}\colon \rho \to (\mathrm{nat} \to \rho \to \rho) \to \mathrm{nat} \to \rho,$$
  $$R_{\mathrm{boole},\rho}\colon \rho \to \rho \to \mathrm{boole} \to \rho$$

  are terms.

We add the following conversion rules (writing $t+1$ for St).

$$R_{\mathrm{nat},\rho} rs0 \to_0 r,$$
$$R_{\mathrm{nat},\rho} rs(t+1) \to_0 st(R_{\mathrm{nat},\rho} rst)$$

and

$$R_{\mathrm{boole},\rho} rs\,\mathrm{true} \to_0 r,$$
$$R_{\mathrm{boole},\rho} rs\,\mathrm{false} \to_0 s.$$

Formulas are built from atomic formulas by means of $\to$, $\wedge$, $\forall$ and $\exists^*$. We let $\bot := \mathrm{atom}(\mathrm{false})$, $\neg A := A \to \bot$ and (writing p for $\mathrm{atom}(p)$) $A \vee^* B := \exists^* p.(p \to A) \wedge (\neg p \to B)$.

We also extend the notion of a derivation term by constants for the truth axiom and induction axioms. Hence derivation terms in arithmetic are obtained by adding the clauses

(T) $\mathrm{T}\colon \mathrm{atom}(\mathrm{true})$ is a derivation term with $\mathrm{FA}(\mathrm{T}) = \emptyset$.

(Ind) For any formula $\forall n\, A$

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

is a derivation term with $\mathrm{FA}(\mathrm{Ind}_{n,A}) = \emptyset$. Similarly, for any formula $\forall p\, A$

$$\mathrm{Ind}_{p,A}\colon \forall . A[\mathrm{true}/p] \to A[\mathrm{false}/p] \to \forall p\, A$$

is a derivation term with $\mathrm{FA}(\mathrm{Ind}_{p,A}) = \emptyset$.

Clearly $\mathrm{FV}(\mathrm{T}) := FV(\mathrm{Ind}_{n,A}) := \mathrm{FV}(\mathrm{Ind}_{p,A}) = \emptyset$.

In the sequel $\vdash$ refers to derivability in the arithmetical system determined by the derivation terms just described.

$\mathrm{Stab}_{\mathrm{atom}}\colon \forall p. \neg\neg \mathrm{atom}(p) \to \mathrm{atom}(p)$ can now be proved easily by boolean induction, using the truth axiom in the case true. As above we can conclude $\vdash \neg\neg A \to A$ for formulas A without $\exists^*$, i.e. built with $\to$, $\wedge$ and $\forall$. We also have $\vdash \bot \to A$ for any A. Furthermore we can derive the usual axioms for $\vee^*$.

The notion of extracted terms can straightforwardly be extended to this situation, and the Soundness Theorem carries over easily. In the case of $\mathrm{Ind}_{n,A}$ we have to prove

$$\mathrm{ets}(\mathrm{Ind}_{n,A})\,\mathrm{mr}\, \forall \vec{x}. A[0/n] \to (\forall n. A \to A[n+1/n]) \to \forall n\, A,$$

i.e.

$$\begin{aligned}\forall \vec{x} \forall \vec{y} \forall \vec{f} \forall n. \vec{y}\,\mathrm{mr}\, A[0/n] \to (\forall n \forall \vec{y}_1. \vec{y}_1\,\mathrm{mr}\, A \to \vec{f} n \vec{y}_1\,\mathrm{mr}\, A[n+1/n])\\ \to \mathrm{ets}(\mathrm{Ind}_{n,A}) \vec{x} \vec{y} \vec{f} n\,\mathrm{mr}\, A.\end{aligned}$$

Hence we let $\mathrm{ets}(\mathrm{Ind}_{n,A}) := \lambda \vec{x}. R_1, \dots, R_k$ where k is the length of $\tau(A) = \rho_1, \dots, \rho_k$ and $R_1, \dots, R_k$ are simultaneous primitive recursion operators of type $R_i\colon \vec{\rho} \to (\mathrm{nat} \to \vec{\rho} \to \vec{\rho}) \to \mathrm{nat} \to \rho_i$ satisfying

$$R_i \vec{y} \vec{f} 0 = y_i,$$
$$R_i \vec{y} \vec{f} (z+1) = f_i z (R_1 \vec{y} \vec{f} z) \dots (R_k \vec{y} \vec{f} z)$$

where $=$ denotes equality of $\beta\eta R$–normal forms. Using these equations we can then prove the above claim easily (recall that terms with the same normal form are identified). The operators $R_1, \dots, R_n$ can be defined easily from the recursion constant $R_{\mathrm{nat}, \rho_1 \times \dots \times \rho_k}$. We could equally well have introduced them as constants and added the equations above as conversion rules.

Boolean induction, i.e. case analysis is treated similarly. We let

$$\mathrm{ets}(\mathrm{Ind}_{p,A}) := \lambda \vec{x}. R_1, \dots, R_k,$$

where now $R_1, \dots, R_k$ are simultaneous primitive recursion (or case splitting) operators of type $R_i \colon \vec{\rho} \to \vec{\rho} \to \mathrm{boole} \to \rho_i$ satisfying

$$R_i \vec{y} \vec{z}\, \mathrm{true} = y_i,$$
$$R_i \vec{y} \vec{z}\, \mathrm{false} = z_i.$$

The lemmata stating that ets commutes with substitution and reduction remain valid since the conversion rules for induction and recursion fit together.

The following remarks and definitions will be helpful later. Let us call a formula A *decidable* if there is a term t_A such that $\vdash A \leftrightarrow \mathrm{atom}(t_A)$.

1. Every quantifier–free formula is decidable. First let $\supset := \lambda p \lambda q. Rq\, \mathrm{true}\, p$ and $\& := \lambda p \lambda q. R(R\, \mathrm{true}\, \mathrm{false}\, q)\, \mathrm{false}\, p$. Clearly

$$\forall p, q. (\mathrm{atom}(p) \to \mathrm{atom}(q)) \leftrightarrow \mathrm{atom}(\supset pq)$$
$$\forall p, q. (\mathrm{atom}(p) \wedge \mathrm{atom}(q)) \leftrightarrow \mathrm{atom}(\& pq)$$

are provable. Hence we let

$$t_{\mathrm{atom}(r)} := r,$$
$$t_{A \to B} := \supset t_A t_B$$
$$t_{A \wedge B} := \& t_A t_B.$$

2. We can do case splitting according to decidable formulas A, i.e. for every formula $B[\vec{x}]$ we can prove

$$\mathrm{Cases}_{A,B} \colon (A \to B) \to (\neg A \to B) \to B.$$

The derivation $\mathrm{Cases}_{A,B}$ is given by

$$\lambda u_1, u_2. \mathrm{Ind}\, \vec{x} (\lambda u_3 \lambda u_4. u_3 \mathrm{T})(\lambda u_5 \lambda u_6. u_6 \neg \mathrm{F}) t_A (\lambda u_7. u_1(d_1 u_7))(\lambda u_8. u_2(d_2 u_8))$$

where $d_1^{\mathrm{atom}(t_A) \to A}$, and $d_2^{\neg \mathrm{atom}(t_A) \to \neg A}$ are derivations which exist according to 1 and the axioms and assumption variables with indices are (writing t for $\mathrm{atom}(t)$)

$$\mathrm{Ind}_{p, (p \to B) \to (\neg p \to B) \to B},$$

$u_1^{A \to B}$, $u_2^{\neg A \to B}$, $u_3^{\mathrm{true} \to B}$, $u_4^{\neg \mathrm{true} \to B}$, $u_5^{\mathrm{false} \to B}$, $u_6^{\neg \mathrm{false} \to B}$, $u_7^{t_A}$ and $u_8^{\neg t_A}$. The extracted terms of $\mathrm{Cases}_{A,B}$ are given by

$$\mathrm{ets}(\mathrm{Cases}_{A,B}) = \lambda \vec{y}, \vec{z}. \mathbf{if}\, t_A \vec{y} \vec{z} =_\eta \mathbf{if}\, t_A$$

where $\mathbf{if} := \vec{R}(\lambda \vec{y}_1, \vec{z}_1. \vec{y}_1)(\lambda \vec{y}_1, \vec{z}_1. \vec{z}_1)$ and $\vec{y}, \vec{z}, \vec{y}_1, \vec{z}_1$ are lists of variables of types $\vec{\rho} := \tau(B)$. Clearly

$$\mathbf{if}\, \mathrm{true}\, \vec{r} \vec{s} =_{\beta R} \vec{r}, \quad \mathbf{if}\, \mathrm{false}\, \vec{r} \vec{s} =_{\beta R} \vec{s}.$$

For better readability we use for $\mathbf{if}\, t_A \vec{r} \vec{s}$ the notation

$$\mathbf{if} \quad A \quad \mathbf{then} \quad \vec{r} \quad \mathbf{else} \quad \vec{s} \quad \mathbf{fi}.$$

2. Classical proofs as programs

As is well known a proof of a $\forall\exists$–theorem with a quantifier–free kernel — where $\exists$ is viewed as defined by $\neg\forall\neg$ — can be used as a program. We describe a "direct method" to use such a proof as a program, and compare it with Harvey Friedman's A–translation [3] followed by the well–known program extraction from constructive proofs.

First note that a classical proof of $\forall x \exists y\, A$ generally does *not* yield a program to compute y from x. The reason for this is that there might be a universal quantifier $\forall z$ right after $\exists y$, i.e. after $\neg\forall y\neg$, and this makes it possible that an assumption

$$\forall y \neg \forall z\, B$$

is instantiated with a non–constant term containing critical variables which are bound later by $\forall z$.

It is well known that this is not just a technical difficulty: if T denotes Kleene's T–predicate, then

$$\forall n \exists m \forall k. T(n,n,k) \to T(n,n,m)$$

is trivially provable even in minimal logic (with $\exists m$ defined as $\neg\forall m\neg$, i.e. in classical logic), but there is no computable function f satisfying

$$\forall n, k. T(n,n,k) \to T(n,n,f(n)),$$

for then $\exists k\, T(n,n,k)$ would be decidable: it would be true if and only if $T(n,n,f(n))$ holds.

Hence in the rest of this section we will only consider formulas of the form $\forall x \exists y\, A$ with A quantifier–free.

We now describe a "direct method" (cf. [13]) to extract the computational content from a classical proof.

By a Π–*formula* we mean a formula built without the strong existential quantifier $\exists^*$, which has no (universal) quantifier in premises of implications. For instance any Horn formula $\forall \vec{x}. P_1(\vec{x}) \to \cdots \to P_n(\vec{x}) \to Q(\vec{x})$ is a Π–formula, but

$$\forall x, y, u. (\forall z. xz = yz) \to ux = uy$$

is not. Clearly every Π–formula is equivalent (in minimal logic) to a conjunction of formulas $\forall C$ where C is quantifier–free and without $\wedge$. So from now on we will assume that Π–formulas are of this form.

A derivation d is called a *refutation of* Π–*assumptions* if d derives a closed false atom from assumptions $\mathrm{FA}(d) = \{v_1\colon \forall C_1, \ldots, v_n\colon \forall C_n\}$ where each C_i is quantifier–free.

Now let d be a refutation of Π–assumptions. We may assume $\mathrm{FV}(d) = \emptyset$ (if not, substitute arbitrary closed terms for the free variables in d). Next we can normalize d. Let $d{\downarrow}$ be the result. Again $d{\downarrow}$ is a refutation of Π–assumptions. We then can read off from $d{\downarrow}$ a list $|d|$ of closed terms called the "first instance" of $d{\downarrow}$ (cf. [13]) such that one of the Π–assumptions is false at $|d|$. To make this notion easier to understand let us restrict the general situation slightly. A closed quantifier–free formula B is *true* respectively *false* if t_B normalizes to the boolean constant true respectively false. A closed Π–formula $\forall\vec{x}\,C$ is true iff for all closed terms $\vec{t}$ the formula $C[\vec{t}/\vec{x}]$ is true.

Let $d{:}\,\bot$ be a normal derivation with $\mathrm{FV}(d)=\emptyset$ of $\bot$ from assumptions

$$\begin{aligned}&u{:}\,\forall\vec{y}.B_1[\vec{y}]\to\ldots\to B_m[\vec{y}]\to\bot,\\&v_1{:}\,\forall C_1,\ldots,v_n{:}\,\forall C_n\end{aligned}$$

where $\forall C_1,\ldots,\forall C_n$ are true closed Π–formulas. We define a list $|d|$ of closed terms, called the *first instance* of d, such that $B_1[|d|],\ldots,B_m[|d|]$ are true. $|d|$ is defined by induction on d. Since d is normal and $\mathrm{FV}(d)=\emptyset$ it does not contain axioms (exept the truth axiom, which is a closed Π–formulas and hence may be assumed to be among the Π–assumptions $\forall C_i$). To see this recall that the normal form of any closed term of type nat is of the form $S(S(S\ldots(S0)\ldots))$ and of any closed term of type boole is either true or false; hence all induction axioms unfold. Therefore d is of the form

$$w\vec{s}d_1\ldots d_k,$$

where $\vec{s}$ are closed terms and $d_1,\ldots,d_k$ are derivations of closed quantifier–free formulas. We distinguish two cases.

1. $d_1,\ldots,d_k$ derive only true formulas (which can be decided, since the formulas are quantifier–free and closed). Then w cannot be one of the v_i since all $\forall C_i$ are true. Hence $d=u\vec{s}d_1\ldots d_k$ and the d_i derive $B_i[\vec{s}]$. So let $|d|:=\vec{s}$.
2. There is a minimal i such that d_i derives a false formula, $A_1\to\cdots\to A_{n_i}\to\bot$ say. Then $A_1\ \ldots,A_{n_i}$ are true. Without loss of generality we may assume that $d_i=\lambda w_1^{A_1}\ldots\lambda w_{n_i}^{A_{n_i}}\,e$ where $e{:}\,\bot$ contains assumptions among

$$\begin{aligned}&u{:}\,\forall\vec{y}.B_1[\vec{y}]\to\ldots\to B_m[\vec{y}]\to\bot,\\&v_1{:}\,\forall C_1,\ldots,v_n{:}\,\forall C_n,\\&w_1{:}\,A_1,\ldots,w_{n_i}{:}\,A_{n_i}.\end{aligned}$$

   Therefore we can recursively define $|d|:=|e|$.

Hence from a proof d of $\forall\vec{x}\exists\vec{y}\bigwedge_i B_i[\vec{x},\vec{y}]$ from true Π–assumptions with B_i quantifier–free we can obtain the following algorithm to compute for any $\vec{r}$ an $\vec{s}$ such that all $B_i[\vec{r},\vec{s}]$ hold. First instanciate d with $\vec{r}$, i.e. form $d\vec{r}{:}\,\exists\vec{y}\bigwedge_i B_i[\vec{r},\vec{y}]$. Since $\exists$ is $\neg\forall\neg$, we have $d\vec{r}u{:}\,\bot$ with $u{:}\,\forall\vec{y}.B_1[\vec{r},\vec{y}]\to\ldots\to B_m[\vec{r},\vec{y}]\to\bot$ a new Π–assumption (of a false formula!). Now normalize $d\vec{r}u$. From its normal form $(d\vec{r}u){\downarrow}$, which is a refutation from Π–assumptions, we can read off the first instance $|(d\vec{r}u){\downarrow}|$ of $(d\vec{r}u){\downarrow}$. These are closed terms $\vec{s}$ such that all $B_i[\vec{r},\vec{s}]$ are true.

It might seem that instead of the method described, which chooses the branch to follow by checking whether some quantifier–free formulas are false or true, one could alternatively look for an occurrence of the false Π–assumption u in the proof whose arguments do not contain u any more. However, in our general case where Π–assumptions (and not just

Horn formulas) are allowed these arguments may contain free assumption variables bound later (by $\to^+$) in the proof, and so we cannot conclude that all arguments of u derive true formulas. If, however, we restrict attention to the special case where we only allow Horn–formulas as assumptions, then this phenomenon cannot happen and we have a variant of the direct method.

Some comments are to be made here.

1. In principle, of course, we could replace any quantifier–free formula C by the atomic formula $\mathrm{atom}(t_C)$. However, this introduces quite a lot of somewhat artificial boolean functions, which makes the proof much harder to read and work with.
2. If we are prepared to apply some preparatory "pruning"–step to our derivation then we may assume that such additional assumptions v are always true. For if we had a $v\colon A$ with A false which is bound later in the proof yielding $A \to C$, then we could replace the whole subproof above this occurrence of $A \to C$ by a derivation of $A \to C$ using ex–falso–quodlibet. If we do this preparatory step first, then the variant of the direct method can be applied to the general case of Π–assumptions as well.

We now describe Friedman's A–translation from [3]. Let A be an arbitrary but fixed formula. The A–translation B^A of a formula B is obtained by replacing any atomic subformula P of B by $(P \to A) \to A$.

Note that any derivation d of some formula B from assumptions $C_1, \dots, C_n$ becomes after the A–translation a derivation of B^A from $C_1^A, , \dots, C_n^A$. To see this recall that our logical rules are those of minimal logic and hence give no extra treatment to falsity. Also the axiom schemes (exept the truth axiom, which can be viewed as a Π–assumption) remain instances of the same axiom scheme after the A–translation. E.g. boolean induction

$$B[\mathrm{true}/p] \to B[\mathrm{false}/p] \to \forall p\, B$$

is translated into

$$B^A[\mathrm{true}/p] \to B^A[\mathrm{false}/p] \to \forall p\, B^A,$$

which again is an instance of boolean induction.

Let us look at what happens with Π–assumptions under the A–translation. As before we may assume that all formulas considered do not contain $\wedge$.

Lemma 1. *For any quantifier–free formula C we can find a derivation $d\colon C \to C^A$.*

Proof by induction on C. Let $C \equiv B_1 \to \dots \to B_m \to R$ with an atom R. We must derive

$$(\vec{B} \to R) \to \vec{B}^A \to (R \to A) \to A.$$

So assume

$$\begin{aligned} \tilde{u}&\colon \vec{B} \to R,\\ \tilde{v}_i&\colon \vec{B}_i^A,\\ w&\colon R \to A. \end{aligned}$$

We must show A.

Case $u_i^-: \neg B_i$ for some i. Let $B_i \equiv \vec{C}_i \to P_i$ with atoms P_i. Then

$$\tilde{v}_i : \vec{C}_i^{\,A} \to (P_i \to A) \to A$$

and we have

$$e_{ij}[u_i^-] :\equiv \mathrm{Stab}_{C_{ij}} \lambda v^{\neg C_{ij}}.u_i^- \lambda \vec{u}^{\vec{C}_i}.\mathrm{Efq}_{P_i}(v u_j) : C_{ij},$$
$$e_i[u_i^-] :\equiv \lambda w_i^{P_i}.u_i^- \lambda \vec{u}^{\vec{C}_i} w_i : \neg P_i.$$

By IH we have $d_{C_{ij}} : C_{ij} \to C_{ij}^A$. Hence

$$\tilde{v}_i(d_{C_{i1}} e_{i1}) \dots (d_{C_{in_i}} e_{in_i})(\lambda w_i^{P_i}.\mathrm{Efq}_A(e_i w_i)) : A.$$

Case $u_i^+ : B_i$ for all i. Then

$$w(\tilde{u} u_1^+ \dots u_m^+) : A$$

The extracted terms for this derivation are

$$\begin{aligned} d^{\mathrm{ets}} \equiv \lambda \vec{x}_1, \dots, \vec{x}_m, \vec{z}.\ & \textbf{if} \quad \neg B_1 \quad \textbf{then} \quad \vec{x}_1 d^{\mathrm{ets}}_{C_{11}} \dots d^{\mathrm{ets}}_{C_{1n_1}} \vec{0} \quad \textbf{else} \\ & \dots \\ & \textbf{if} \quad \neg B_m \quad \textbf{then} \quad \vec{x}_m d^{\mathrm{ets}}_{C_{m1}} \dots d^{\mathrm{ets}}_{C_{mn_m}} \vec{0} \quad \textbf{else} \\ & \vec{z} \quad \textbf{fi} \dots \textbf{fi}, \end{aligned}$$

where $\vec{x}_i, \vec{z}$ are the lists of variables associated with $\tilde{v}_i : B_i^A, w : R \to A$. □

Here we have used case splitting according to the quantifier free formulas B_i which is admissible by a remark above.

If we want to use the A–translation to extract the computational content from a classical proof we have to choose a particular A involving the strong existential quantifier.

Lemma 2. *Let $B_i[\vec{x}, \vec{y}]$ be quantifier–free formulas and $A[\vec{x}] := \exists^* \vec{y} \bigwedge_i B_i[\vec{x}, \vec{y}]$. Then we can find a derivation of $(\forall \vec{y}.B_1[\vec{x}, \vec{y}] \to \dots \to B_m[\vec{x}, \vec{y}] \to \bot)^{A[\vec{x}]}$.*

Proof. Let $\vec{y}$ be given and assume $\tilde{v}_i : \vec{B}_i^A$ and $w : \bot \to A$. We must show A.

Case $u_i^- : \neg B_i$ for some i. Let $B_i \equiv \vec{C}_i \to P_i$ with atoms P_i. Then

$$\tilde{v}_i : \vec{C}_i^{\,A} \to (P_i \to A) \to A$$

and we have

$$e_{ij}[u_i^-] :\equiv \mathrm{Stab}_{C_{ij}} \lambda v^{\neg C_{ij}}.u_i^- \lambda \vec{u}^{\vec{C}_i}.\mathrm{Efq}_{P_i}(v u_j) : C_{ij},$$
$$e_i[u_i^-] :\equiv \lambda w_i^{P_i}.u_i^- \lambda \vec{u}^{\vec{C}_i} w_i : \neg P_i.$$

Using $d_{C_{ij}} : C_{ij} \to C_{ij}^A$ from Lemma 1 we obtain

$$\tilde{v}_i(d_{C_{i1}} e_{i1}) \dots (d_{C_{in_i}} e_{in_i})(\lambda w_i^{P_i}.\mathrm{Efq}_A(e_i w_i)) : A.$$

Case $u_i^+ : B_i$ for all i. Then

$$\exists^+ \vec{y} \langle u_1^+, \dots, u_m^+ \rangle : A$$

The extracted terms for this derivation are

$$\begin{aligned} d^{\mathrm{ets}} \equiv \lambda \vec{y}, \vec{x}_1, \dots, \vec{x}_m, \vec{z}.\ & \mathbf{if} \quad \neg B_1 \quad \mathbf{then} \quad \vec{x}_1 d^{\mathrm{ets}}_{C_{11}} \dots d^{\mathrm{ets}}_{C_{1n_1}} \vec{0} \quad \mathbf{else} \\ & \dots \\ & \mathbf{if} \quad \neg B_m \quad \mathbf{then} \quad \vec{x}_m d^{\mathrm{ets}}_{C_{m1}} \dots d^{\mathrm{ets}}_{C_{mn_m}} \vec{0} \quad \mathbf{else} \\ & \vec{y} \quad \mathbf{fi} \dots \mathbf{fi}, \end{aligned}$$

where $\vec{x}_i, \vec{z}$ are the lists of variables associated with $\tilde{v}_i : B_i^A, w : \bot \to A$. □

Theorem (Friedman). *For any derivation*

$$d[u : \forall \vec{y}. B_1[\vec{x}, \vec{y}] \to \dots \to B_m[\vec{x}, \vec{y}] \to \bot, v_1 : \forall C_1, \dots, v_n : \forall C_n] : \bot$$

with B_i, C_j *quantifier–free we can find a derivation*

$$d^{\mathrm{tr}}[v_1 : \forall C_1, \dots, v_n : \forall C_n] : \exists^* \vec{y} \bigwedge_i B_i[\vec{x}, \vec{y}].$$

Proof. Let $A[\vec{x}] := \exists^* \vec{y} \bigwedge_i B_i[\vec{x}, \vec{y}]$ and consider the $A[\vec{x}]$–translation

$$\begin{aligned} d^{A[\vec{x}]}&[u' : (\forall \vec{y}. B_1[\vec{x}, \vec{y}] \to \dots \to B_m[\vec{x}, \vec{y}] \to \bot)^{A[\vec{x}]}, v_1' : (\forall C_1)^{A[\vec{x}]}, \dots v_n' : (\forall C_n)^{A[\vec{x}]}] \\ &: (\bot \to A[\vec{x}]) \to A[\vec{x}] \end{aligned}$$

of d, obtained by just changing some formulas. By Lemma 1 we have

$$d_{v_i}[v_i : \forall C_i] : (\forall C_i)^{A[\vec{x}]}.$$

By Lemma 2 (now using the particular choice of $A[\vec{x}]$) the $A[\vec{x}]$–translation of the assumption u is provable without assumptions:

$$d_u : (\forall \vec{y}. B_1[\vec{x}, \vec{y}] \to \dots \to B_m[\vec{x}, \vec{y}] \to \bot)^{A[\vec{x}]}$$

Substituting $d_{v_i}[v_i : \forall C_i]$ for v_i' and d_u for u' we obtain

$$d^{\mathrm{tr}}[v_1 : \forall C_1, \dots, v_n : \forall C_n] : \exists^* \vec{y} \bigwedge_i B_i[\vec{x}, \vec{y}],$$

where

$$d^{\mathrm{tr}} \equiv d^{A[\vec{x}]}[d_u, d_{v_1}, \dots, d_{v_n}] \mathrm{Efq}_{A[\vec{x}]}.$$ □

Having obtained a proof d^{tr} of an existential formula $\exists^* \vec{y} \bigwedge_i B_i[\vec{x}, \vec{y}]$ we can then apply the general method of extracting terms to this proof. It yields

$$(d^{\mathrm{tr}})^{\mathrm{ets}} \equiv (d^{A[\vec{x}]})^{\mathrm{ets}}[d_u^{\mathrm{ets}}, d_{v_1}^{\mathrm{ets}}, \dots, d_{v_n}^{\mathrm{ets}}] \vec{0}, \tag{1}$$

since extracting terms commutes with substitution.

Note that there are many ways to use Friedman's method to extract a term from a given arithmetical proof of a weak existential formula.

1. The most straightforward way is to prove the formula without any assumptions. This means that we are not allowed to use lemmata, and hence that such a proof tends to be rather long, and difficult to produce.
2. The next straightforward way is to pack all Π–Lemmata used in the proof (and proved explicitely in 1 above) into purely generalized atoms $\forall\vec{x}\,\mathrm{atom}(t)$. However, this means that we have to introduce rather complex boolean terms, which will later show up in the proof as tests for case distinctions.

We now prove that the value of the extracted terms $(d^{\mathrm{tr}})^{\mathrm{ets}}$ when instanciated with a list $\vec{r}$ of closed terms is in fact the same as the result of the direct method described above.

So consider again the situation of Friedman's Theorem, i.e. a derivation

$$d[u\colon \forall\vec{y}.B_1[\vec{x},\vec{y}] \to \ldots \to B_m[\vec{x},\vec{y}] \to \bot, v_1\colon \forall C_1,\ldots,v_n\colon \forall C_n]\colon \bot$$

with B_i, C_j quantifier–free. We just observed that the program $(d^{\mathrm{tr}})^{\mathrm{ets}}$ extracted from the translated derivation has the form (1) above. Let us try to understand how this program works. First, $(d^{A[\vec{x}]})^{\mathrm{ets}}$ closely follows the structure of d. The reason is that $d^{A[\vec{x}]}$ differs from d only with respect to the formulas affixed, and when forming the extracted terms this affects only the types and the arities of the lists of object variables associated with assumption variables.

In order to comprehend $d_{v_i}^{\mathrm{ets}}$ and d_u^{ets} let us have a second look at the proofs of Lemma 1 and 2. First note that $d_{v_i}[v_i\colon \forall C_i]\colon (\forall C_i)^{A[\vec{x}]}$ is obtained from $d_i\colon C_i \to C_i^{A[\vec{x}]}$ constructed in the proof of Lemma 1 by

$$d_{v_i} \equiv \lambda\vec{y}_i.d_i(v_i\vec{y}_i).$$

Since v_i has type $\forall C_i$, which is a Harrop formula, we have $d_{v_i}^{\mathrm{ets}} \equiv \lambda\vec{y}_i\, d_i^{\mathrm{ets}}$. Now from the proof of Lemma 1 we obtain

$$\begin{aligned} d_i^{\mathrm{ets}} \equiv \lambda\vec{x}_1,\ldots,\vec{x}_m,\vec{z}.\ &\mathbf{if}\quad \neg B_1\quad \mathbf{then}\quad \vec{x}_1 d_{C_{11}}^{\mathrm{ets}}\ldots d_{C_{1n_1}}^{\mathrm{ets}}\vec{0}\quad \mathbf{else}\\ &\ldots\\ &\mathbf{if}\quad \neg B_m\quad \mathbf{then}\quad \vec{x}_m d_{C_{m1}}^{\mathrm{ets}}\ldots d_{C_{mn_m}}^{\mathrm{ets}}\vec{0}\quad \mathbf{else}\\ &\vec{z}\quad \mathbf{fi}\ldots\mathbf{fi}, \end{aligned} \tag{2}$$

where $C_i \equiv B_1 \to \ldots \to B_m \to R$ with $B_i \equiv \vec{C}_i \to P_i$ and $\vec{x}_i, \vec{z}$ are the lists of variables associated with $\tilde{v}_i, w$. Furthermore, $d_{C_{ij}}^{\mathrm{ets}}$ are the extracted terms of derivations $d_{C_{ij}}\colon C_{ij} \to C_{ij}^{A[\vec{x}]}$ constructed by previous applications of Lemma 1. Similarly

$$\begin{aligned} d_u^{\mathrm{ets}} \equiv \lambda\vec{y},\vec{x}_1,\ldots,\vec{x}_m,\vec{z}.\ &\mathbf{if}\quad \neg B_1\quad \mathbf{then}\quad \vec{x}_1 d_{C_{11}}^{\mathrm{ets}}\ldots d_{C_{1n_1}}^{\mathrm{ets}}\vec{0}\quad \mathbf{else}\\ &\ldots\\ &\mathbf{if}\quad \neg B_m\quad \mathbf{then}\quad \vec{x}_m d_{C_{m1}}^{\mathrm{ets}}\ldots d_{C_{mn_m}}^{\mathrm{ets}}\vec{0}\quad \mathbf{else}\\ &\vec{y}\quad \mathbf{fi}\ldots\mathbf{fi}, \end{aligned} \tag{3}$$

where $B_i \equiv \vec{C}_i \to P_i$ and $\vec{x}_i, \vec{z}$ are the lists of variables associated with $\tilde{v}_i, w$. Furthermore, $d_{C_{ij}}^{\mathrm{ets}}$ are the extracted terms of derivations $d_{C_{ij}}\colon C_{ij} \to C_{ij}^{A[\vec{x}]}$ constructed by previous applications of Lemma 1.

This analysis makes it possible to prove that module $\beta\eta R$ the extracted terms when instanciated with a list $\vec{r}$ of closed terms are in fact the same as the result of the direct method described above to read off the first instance provided by the instanciated derivation

$$\bar{d}[\bar{u}\colon \forall \vec{y}.B_1[\vec{r},\vec{y}] \to \ldots \to B_m[\vec{r},\vec{y}] \to \bot, v_1\colon \forall C_1, \ldots, v_n\colon \forall C_n]\colon \bot$$

Below we will show the following

Claim. *For any normal derivation*

$$e[\bar{u}\colon \forall \vec{y}.B_1[\vec{r},\vec{y}] \to \ldots \to B_m[\vec{r},\vec{y}] \to \bot, v_1\colon \forall C_1, \ldots, v_n\colon \forall C_n]\colon \bot$$

with $\mathrm{FV}(e) = \emptyset$ *we have*

$$|e| =_{\beta\eta R} (e^{A[\vec{r}]})^{\mathrm{ets}}[d_u^{\mathrm{ets}}[\vec{r}/\vec{x}], d_{v_1}^{\mathrm{ets}}, \ldots, d_{v_n}^{\mathrm{ets}}]\vec{0}.$$

We then obtain that the instanciation of the extracted terms (1) with $\vec{r}$ for $\vec{x}$, i.e.

$$(d^{A[\vec{x}]})^{\mathrm{ets}}[d_u^{\mathrm{ets}}, d_{v_1}^{\mathrm{ets}}, \ldots, d_{v_n}^{\mathrm{ets}}]\vec{0}[\vec{r}/\vec{x}] \equiv (d[\vec{r}/\vec{x}]^{A[\vec{r}]})^{\mathrm{ets}}[d_u^{\mathrm{ets}}[\vec{r}/\vec{x}], d_{v_1}^{\mathrm{ets}}, \ldots, d_{v_n}^{\mathrm{ets}}]\vec{0}$$

has as its value the list of closed terms which is the first instance of the instanciated derivation $d[\vec{r}/\vec{x}]$, i.e. $|d[\vec{r}/\vec{x}]{\downarrow}|$. For by the claim we have

$$\begin{aligned}
|d[\vec{r}/\vec{x}]{\downarrow}| &=_{\beta\eta R} ((d[\vec{r}/\vec{x}]{\downarrow})^{A[\vec{r}]})^{\mathrm{ets}}[d_u^{\mathrm{ets}}[\vec{r}/\vec{x}], d_{v_1}^{\mathrm{ets}}, \ldots, d_{v_n}^{\mathrm{ets}}]\vec{0} \\
&=_{\beta\eta R} ((d[\vec{r}/\vec{x}]^{A[\vec{r}]})^{\mathrm{ets}}){\downarrow}[d_u^{\mathrm{ets}}[\vec{r}/\vec{x}], d_{v_1}^{\mathrm{ets}}, \ldots, d_{v_n}^{\mathrm{ets}}]\vec{0} \\
&=_{\beta\eta R} (d[\vec{r}/\vec{x}]^{A[\vec{r}]})^{\mathrm{ets}}[d_u^{\mathrm{ets}}[\vec{r}/\vec{x}], d_{v_1}^{\mathrm{ets}}, \ldots, d_{v_n}^{\mathrm{ets}}]\vec{0},
\end{aligned}$$

since normalization (i.e. $\beta\eta\exists^* R$–conversion) commutes with $A[\vec{r}]$–translation and the formation of extracted terms.

It remains to prove the claim. We use induction on e. Since e is normal, it must be of the form $e = w\vec{s}e_1 \ldots e_k$ with $w \in \{\bar{u}, v_1, \ldots, v_n\}$.

Case 1. $e_1, \ldots, e_k$ derive only true formulas. Then $w = \bar{u}$, $k = m$ and the e_i derive $B_i[\vec{r},\vec{s}]$. By definition $|e| := \vec{s}$. Furthermore

$$\begin{aligned}
&(e^{A[\vec{r}]})^{\mathrm{ets}}[d_u^{\mathrm{ets}}[\vec{r}/\vec{x}], d_{v_1}^{\mathrm{ets}}, \ldots, d_{v_n}^{\mathrm{ets}}]\vec{0} \\
&\equiv d_u^{\mathrm{ets}}[\vec{r}/\vec{x}]\vec{s}(e_1^{A[\vec{r}]})^{\mathrm{ets}}[d_u^{\mathrm{ets}}[\vec{r}/\vec{x}], d_{v_1}^{\mathrm{ets}}, \ldots, d_{v_n}^{\mathrm{ets}}] \ldots (e_m^{A[\vec{r}]})^{\mathrm{ets}}[d_u^{\mathrm{ets}}[\vec{r}/\vec{x}], d_{v_1}^{\mathrm{ets}}, \ldots, d_{v_n}^{\mathrm{ets}}]\vec{0} \\
&=_{\beta R} \vec{s}
\end{aligned}$$

by the form (3) of d_u^{ets}, since all $B_i[\vec{r},\vec{s}] \equiv (\vec{B}_i \to P_i)[\vec{r},\vec{s}]$ are true.

Case 2. There is a minimal i such that e_i derives a false formula, $D_{i1}[\vec{s}] \to \cdots \to D_{in_i}[\vec{s}] \to \bot$ say. Then $D_{i1}[\vec{s}], \dots, D_{in_i}[\vec{s}]$ are true. Without loss of generality we may assume that $e_i = \lambda w_1^{D_{i1}[\vec{s}]} \dots \lambda w_m^{D_{in_i}[\vec{s}]} f$ where $f\colon \bot$ contains assumptions among

$$\begin{aligned}
&\bar{u}\colon \forall \vec{y}. B_1[\vec{r}, \vec{y}] \to \ldots \to B_m[\vec{r}, \vec{y}] \to \bot,\\
&v_1\colon \forall C_1, \dots, v_n\colon \forall C_n,\\
&w_1\colon D_{i1}, \dots, w_m\colon D_{in_i}.
\end{aligned}$$

Therefore by definition $|e| := |f|$. Furthermore using the notation $d_{w_j}[w_j]\colon D_{ij}^{A[\vec{r}]}$ for the derivation obtained by applying Lemma 1 to D_{ij} we have

$$\begin{aligned}
&(e^{A[\vec{r}]})^{\mathrm{ets}}[d_u^{\mathrm{ets}}[\vec{r}], d_{v_1}^{\mathrm{ets}}, \dots, d_{v_n}^{\mathrm{ets}}]\vec{0}\\
&\equiv \begin{cases} d_u^{\mathrm{ets}}[\vec{r}]\vec{s}\,(e_1^{A[\vec{r}]})^{\mathrm{ets}}[d_u^{\mathrm{ets}}[\vec{r}], d_{v_1}^{\mathrm{ets}}, \dots, d_{v_n}^{\mathrm{ets}}] \dots (e_k^{A[\vec{r}]})^{\mathrm{ets}}[d_u^{\mathrm{ets}}[\vec{r}], d_{v_1}^{\mathrm{ets}}, \dots, d_{v_n}^{\mathrm{ets}}]\vec{0} & \text{if } w = \bar{u}\\ d_{v_i}^{\mathrm{ets}}\vec{s}\,(e_1^{A[\vec{r}]})^{\mathrm{ets}}[d_u^{\mathrm{ets}}[\vec{r}], d_{v_1}^{\mathrm{ets}}, \dots, d_{v_n}^{\mathrm{ets}}] \dots (e_k^{A[\vec{r}]})^{\mathrm{ets}}[d_u^{\mathrm{ets}}[\vec{r}], d_{v_1}^{\mathrm{ets}}, \dots, d_{v_n}^{\mathrm{ets}}]\vec{0} & \text{if } w = v_i \end{cases}\\
&=_{\beta R} (e_i^{A[\vec{r}]})^{\mathrm{ets}}[d_u^{\mathrm{ets}}[\vec{r}], d_{v_1}^{\mathrm{ets}}, \dots, d_{v_n}^{\mathrm{ets}}] d_{w_1}^{\mathrm{ets}} \dots d_{w_{n_i}}^{\mathrm{ets}}\,\vec{0} \quad \text{by (3) and (2), respectively}\\
&=_{\beta} (f^{A[\vec{r}]})^{\mathrm{ets}}[d_u^{\mathrm{ets}}[\vec{r}], d_{v_1}^{\mathrm{ets}}, \dots, d_{v_n}^{\mathrm{ets}}, d_{w_1}^{\mathrm{ets}}, \dots, d_{w_{n_i}}^{\mathrm{ets}}]\vec{0},
\end{aligned}$$

so the claim follows from the IH. □

In applications it will be important to produce extracted terms with as few as possible case distinctions, and also that the case distinctions should be over as simple as possible boolean terms. The following example will show that such improvements are indeed necessary.

Let $f\colon \mathrm{nat} \to \mathrm{nat}$ be an unbounded function with $f(0) = 0$. Then we can prove

$$\forall n \exists m. f(m) \le n < f(m+1).$$

If e.g. $f(m) = m^2$, then this formula expresses the existence of an integer square root $m := [\sqrt{n}]$ for any n. More formally we can prove

$$\forall n \exists m. \neg n < f(m) \wedge n < f(m+1) \tag{1}$$

from the assumptions

$$v_1\colon \forall n\, \neg n < f(0), \qquad v_2\colon \forall n\, n < f(g(n)).$$

Here $<^{\mathrm{nat} \to \mathrm{nat} \to \mathrm{boole}}$ is the characteristic function of the natural ordering of the natural numbers and $r < s$ denotes $\mathrm{atom}(<rs)$. We expressed $f(m) \le n$ by $\neg n < f(m)$ and $f(0) = 0$ by $\forall n\, \neg n < f(0)$ to keep the formal proof as simple as possible. In order to have Π–assumptions we had to express the unboundedness of f by a witnessing function g. First note that the Π–assumptions v_1 and v_2 are not closed as required previously. But this is no problem since A–translation and program extraction clearly also work in this case: we just get a program containing f and g as parameters.

Now let us prove (1). Let n be given and assume

$$u\colon \forall m.\neg n < f(m) \to n < f(m+1) \to \bot.$$

We have to show $\bot$. From v_1 and u we inductively get $\forall m\, \neg n < f(m)$. For $m := g(n)$ this yields a contradiction to v_2.

The derivation term corresponding to this proof is

$$d := \mathrm{Ind}_{m,\neg n<f(m)} n(v_1 n)u(g(n))(v_2 n)\colon \bot.$$

Now let

$$A := \exists^* m.\neg n < f(m) \wedge n < f(m+1).$$

The program extracted from d is

$$(d^{\mathrm{tr}})^{\mathrm{ets}} \equiv (d^A)^{\mathrm{ets}}[d_u^{\mathrm{ets}}, d_{v_1}^{\mathrm{ets}}, d_{v_2}^{\mathrm{ets}}]0\colon \mathrm{nat}.$$

$(d^A)^{\mathrm{ets}}$ is the same as d exept that $\mathrm{Ind}_{m,\neg n<f(m)}$ has to be replaced by

$$\lambda n\, R_{\mathrm{nat},(\mathrm{nat}\to\mathrm{nat})\to\mathrm{nat}\to\mathrm{nat}}$$

(since $\tau((\neg n < f(m))^A) = \tau(((n < f(m) \to A) \to A) \to (\bot \to A) \to A) = (\mathrm{nat} \to \mathrm{nat}) \to \mathrm{nat} \to \mathrm{nat}$), and the assumption variables u, v_1, v_2 in d have to be replaced by (unary lists of) object variables

$$\begin{aligned}
&x_u\colon \mathrm{nat} \to [(\mathrm{nat} \to \mathrm{nat}) \to \mathrm{nat} \to \mathrm{nat}] \to (\mathrm{nat} \to \mathrm{nat}) \to \mathrm{nat} \to \mathrm{nat},\\
&x_{v_1}\colon \mathrm{nat} \to (\mathrm{nat} \to \mathrm{nat}) \to \mathrm{nat} \to \mathrm{nat},\\
&x_{v_2}\colon \mathrm{nat} \to \mathrm{nat} \to \mathrm{nat}.
\end{aligned}$$

The subprograms $d_u^{\mathrm{ets}}, d_{v_1}^{\mathrm{ets}}, d_{v_2}^{\mathrm{ets}}$ (of the same types as x_u, x_{v_1}, x_{v_2}) are given by

$$\begin{aligned}
d_u^{\mathrm{ets}} \equiv \lambda m, x_1, x_2, k.\ &\mathbf{if}\quad n < f(m) \quad \mathbf{then} \quad x_1\mathrm{id}0 \quad \mathbf{else}\\
&\mathbf{if}\quad \neg n < f(m+1) \quad \mathbf{then} \quad x_2 0 \quad \mathbf{else}\\
&m \quad \mathbf{fifi},\\
d_{v_1}^{\mathrm{ets}} \equiv \lambda n, x_1, k.\ &\mathbf{if}\quad \neg n < f(0) \quad \mathbf{then} \quad x_1 0 \quad \mathbf{else}\\
&k \quad \mathbf{fi},\\
d_{v_2}^{\mathrm{ets}} \equiv \lambda n, k\, k&.
\end{aligned}$$

Hence the normal form of $(d^{\mathrm{tr}})^{\mathrm{ets}}$ is

$$\begin{aligned}
&R(\lambda x_1, k.\ \mathbf{if}\quad \neg n < f(0) \quad \mathbf{then} \quad x_1 0 \quad \mathbf{else} \quad k \quad \mathbf{fi})\\
&\quad (\lambda m, w_1, w_2, k.\ \mathbf{if}\quad n < f(m) \quad \mathbf{then} \quad w_1\mathrm{id}0 \quad \mathbf{else}\\
&\qquad\qquad\qquad\quad \mathbf{if}\quad \neg n < f(m+1) \quad \mathbf{then} \quad w_2 0 \quad \mathbf{else}\\
&\qquad\qquad\qquad\quad m \quad \mathbf{fifi})\\
&\quad (g(n))\\
&\quad (\lambda k\, k)\\
&\quad 0
\end{aligned}$$

Informally, $(d^{\mathrm{tr}})^{\mathrm{ets}} = H(g(n), \lambda k\, k, 0)$ where $H\colon \mathrm{nat} \to (\mathrm{nat} \to \mathrm{nat}) \to \mathrm{nat} \to \mathrm{nat}$ is such that

$$\begin{aligned} H(0, x_1, k) &= \textbf{if} \quad \neg n < f(0) \quad \textbf{then} \quad x_1 0 \quad \textbf{else} \quad k \quad \textbf{fi} \\ H(m+1, x_1, k) &= \textbf{if} \quad n < f(m) \quad \textbf{then} \quad H(m, \mathrm{id}, 0) \quad \textbf{else} \\ & \ \textbf{if} \quad \neg n < f(m+1) \quad \textbf{then} \quad x_1 0 \quad \textbf{else} \\ & \ m \quad \textbf{fifi} \end{aligned}$$

This program is correct, but it is unnecessarily complicated. However, one can define a refined A–translation which simplifies the type of the auxiliary function H as well as its if–then–else structure. The type reduction can be achieved by *not* replacing *all* atoms P by $(P \to A) \to A$; this of course requires that atom is not the only relation symbol. Furthermore, to reduce case splitting one can construct "better" proofs of $C \to C^A$ for quantifier–free C without $\wedge$. In our example these refinements produce the following simplified progam:

$$\begin{aligned} &h(0) = 0, \\ &h(m+1) = \textbf{if} \quad n < f(m) \quad \textbf{then} \quad h(m) \quad \textbf{else} \quad m \quad \textbf{fi}. \end{aligned}$$

However, space does not permit us to carry the details of the refinements here.

3. Pruning proof trees

It is shown how a proof can be simplified after addition of some further assumptions. In a first step some subproofs are replaced by different ones which use the additional assumptions. In a second step parts of the proof tree are pruned, i.e. cut out. Note that the first step involves searching for new proofs — using the new assumptions — of formulas in the proof tree. Hence we also have to discuss methods for proof search in minimal logic.

Let us describe both pruning steps in more detail.

For the first pruning step let a derivation d be given. For simplicity let us suppose that d contains no free assumption variables, i.e. $\mathrm{FA}(d) = \emptyset$. Assume also that some additional assumptions $v_1\colon B_1, \dots, v_m\colon B_m$ are given. We then go through all nodes of this proof. At any node, we have

- the formula A derived at this node, and
- the assumptions $u_1\colon A_1, \dots, u_n\colon A_n$ active at this node; they are bound further down in the proof by $\to^+$, with u_1 bound above u_2 bound above u_3 and so on.

Then we search for a new proof of A, using the additional assumptions $v_1\colon B_1, \dots, v_m\colon B_m$ and a subset $\{u_1\colon A_1, \dots, u_i\colon A_i\}$ of the active assumptions with i minimal. The reason for giving preference to cutting out assumptions bound further down (i.e. more towards the root) in the proof is that the second pruning step described below then has a chance to prune out greater parts of the proof.

Clearly some systematic way to search for proofs in minimal logic is necessary here. In our implementation of the pruning method we have chosen a variant of Dale Miller's higher order logic programming framework in what he calls the $\mathcal{L}_\lambda$–fragment of our language $\mathcal{L}$; see [7]. It is based on an unification algorithm also given by Miller in [7] for $\mathcal{L}_\lambda$–terms (they are called higher–order patterns in Nipkow [10]). The reason for the restriction to the $\mathcal{L}_\lambda$–fragment is that the unification problem is decidable in this case. It should be noted that any first–order formula is in $\mathcal{L}_\lambda$. — Other interesting approaches to proof search in minimal logic can be found in [2, 6, 12, 14, 15].

We now describe the second pruning step. Let a derivation of a formula A be given. Assume that it contains a proper subderivation e of the same formula A. Assume also that between the endformula A of e and the global endformula A no $\to^+$ rule appears which binds an assumption variable that actually occurs free in e. More precisely, we assume that the derivation term is of the form

$$d^A[e^A/u^A].$$

We then "prune" the proof tree and replace it by its subtree e. This yields a correct derivation, since the subderivation contains no free object variables or assumption variables which have been bound in the original derivation. More precisely, we replace

$$d^A[e^A/u^A] \qquad \text{by} \qquad e^A.$$

Note that this pruning process is not unique: there may be many different occurrences of such subderivations of A. More formally, we may have

$$d^A[e^A/u^A] \equiv d^A[e'^A/v^A].$$

Note also that this is a generalization of Prawitz's immediate simplifications from [11].

The second pruning step then consists in applying the procedure just described to any subderivation of the given derivation.

Example: Proof by cases.

$$\dfrac{\dfrac{(A\to B)\to(\neg A\to B)\to B \qquad \dfrac{\begin{matrix}u\colon A\\ \mid \\ B\end{matrix}}{A\to B}\,(\to^+ u)}{(\neg A\to B)\to B} \qquad \dfrac{\begin{matrix}v\colon \neg A\\ \mid \\ B\end{matrix}}{\neg A\to B}\,(\to^+ v)}{B}$$

More precisely, we have the derivation term

$$\left[(\mathrm{Cases}_{A,B})^{(A\to B)\to(\neg A\to B)\to B}(\lambda u^A\, d_1^B)^{A\to B}(\lambda v^{\neg A}\, d_2^B)^{\neg A\to B}\right]^B.$$

Now assume that e.g. $\neg A$ can be proved from some additional assumptions. Let $e^{\neg A}$ be such a derivation term. Then we can replace the assumption $v\colon \neg A$ by this derivation $e^{\neg A}$:

$$\dfrac{\dfrac{(A\to B)\to(\neg A\to B)\to B \qquad \dfrac{\begin{matrix}u\colon A\\ \mid \\ B\end{matrix}}{A\to B}\,(\to^+ u)}{(\neg A\to B)\to B} \qquad \dfrac{\begin{matrix}\mid\\ \neg A\\ \mid \\ B\end{matrix}}{\neg A\to B}\,(\to^+ v)}{B}$$

or written as a derivation term

$$\left[(\mathrm{Cases}_{A,B})^{(A\to B)\to(\neg A\to B)\to B}(\lambda u^A\, d_1^B)^{A\to B}(\lambda v^{\neg A}\, d_2^B[e^{\neg A}/v^{\neg A}])^{\neg A\to B}\right]^B.$$

Now since $v^{\neg A}$ does not occur free in $e^{\neg A}$, i.e. $v^{\neg A}\notin \mathrm{FA}(e^{\neg A})$, we can write this derivation term as

$$\left[(\mathrm{Cases}_{A,B})^{(A\to B)\to(\neg A\to B)\to B}(\lambda u^A\, d_1^B)^{A\to B}(\lambda v^{\neg A}\, w^B)^{\neg A\to B}\right]^B [d_2^B[e^{\neg A}/v^{\neg A}]/w^B]$$

and then prune it to obtain

$$d_2^B[e^{\neg A}/v^{\neg A}].$$

Note that by this second pruning step a β–normal derivation may be turned into a non–β–normal one. Moreover it can happen that after forming the β–normal form again the resulting derivation once more has a subderivation reducible by the second pruning step. In fact, the following example (due to Jan Ekman) shows that one may get back to the original derivation, and hence one has non–termination. The example only uses minimal implicational logic.

$$((p \to q) \to p) \to (p \to p \to q) \to q$$

(which corresponds to $\neg(p \leftrightarrow \neg p)$ if q is viewed as falsity) is derivable by

$$
\begin{prooftree}
\AxiomC{$p \to p \to q$}
\AxiomC{$(p \to q) \to p$}
\AxiomC{$p \to p \to q$}
\AxiomC{p}
\BinaryInfC{$p \to q$}
\AxiomC{p}
\BinaryInfC{q}
\UnaryInfC{$p \to q$}
\BinaryInfC{p}
\BinaryInfC{$p \to q$}
\AxiomC{$(p \to q) \to p$}
\AxiomC{$p \to p \to q$}
\AxiomC{p}
\BinaryInfC{$p \to q$}
\AxiomC{p}
\BinaryInfC{q}
\UnaryInfC{$p \to q$}
\BinaryInfC{p}
\BinaryInfC{q}
\end{prooftree}
$$

The derivation term is

$$v_1(v_0(\lambda w.v_1 ww))(v_0(\lambda w.v_1 ww))$$

with free assumption variables $v_0\colon (p \to q) \to p$ and $v_1\colon p \to p \to q$, and a bound assumption variable $w\colon p$. This derivation term can be written in the form

$$d^{p\to q}[e^{p\to q}/u^{p\to q}](v_0(\lambda w.v_1 ww)).$$

Now if we let

$$\begin{aligned} d^{p\to q} &:= v_1[v_0 u^{p\to q}], \\ e^{p\to q} &:= \lambda w.v_1 ww, \end{aligned}$$

then replacing the subderivation $d^{p\to q}[e^{p\to q}/u^{p\to q}]$ by $e^{p\to q}$ yields

$$(\lambda w.v_1 ww)\,[v_0(\lambda w.v_1 ww)]\,,$$

which β–converts into the original derivation. We therefore restrict the second pruning step to cases where no new β–redexes are created.

4. An example of program development by pruning

The ideas discussed in the previous section are applied to an example already treated by Goad in his thesis [5], the binpacking problem. The main difference to Goad's work is that he used a logic with the strong existential quantifier, whereas we work within the $\rightarrow\forall$–fragment. This example is particularly well–suited to demonstrate that the pruning method can be applied to adapt programs to particular situations, and moreover that pruning can change the functions computed by programs. In this sense this method is essentially different from program development by program transformation.

The one–dimensional *binpacking problem* can be described as follows. We are given

- p one–dimensional blocks whose size is determined by natural numbers $i_0, \ldots, i_{p-1}$, and
- q one–dimensional bins whose size is determined by natural numbers $j_0, \ldots, j_{q-1}$.

The problem is to decide whether it is possible to pack the blocks into the bins, and if so, to give a solution.

We will treat the obvious first–fit algorithm to solve this problem: First try to pack i_0 into j_0. If it fits, try to pack the remaining blocks $i_1, \ldots, i_{p-1}$ into the modified bins $j_0 - i_0, j_1, \ldots, j_{q-1}$. If they do not fit, try to pack i_0 into the next bin j_1, and so on.

Now consider the particular situation where we know more about the input data, e.g. some order relations between the blocks to be packed and/or between the bins. Then it might be possible to adapt the general algorithm to this particular situation, and it is at least conceivable that the new algorithm which works for our particular situation is even extensionally different from the original one. We will see that this actually can happen, and that one can apply general proof–theoretic machinery to obtain the new, specialized algorithm from the general one.

To apply proof–theoretic methods we first have to construct a proof to work with. This proof should correspond as closely as possible to the general binpacking algorithm described above.

First note that it will be useful to refine the binpacking problem slightly: We also give as an input a natural number m and then look for a solution under the additional condition that the first block i_0 is to be packed in one of the first m bins $j_0, \ldots, j_{m-1}$. Let us call this variant the *bounded binpacking problem*.

Now a proof must be a proof of something, so we must start by giving a formal specification of the bounded binpacking problem. As is well known, there are many choices at this point. In particular, we have to fix a language and the basic properties of the functions and predicates in that language.

We use the two ground types nat for natural numbers and seq for finite sequences of natural numbers. i, j, k, m, n are variables for natural numbers, and a, b, c, x are variables for finite sequences of natural numbers. The empty sequence is denoted by `nil`, and `i::x` denotes the result of constructing a new sequence by adding the number `i` in front of the sequence `x`. The intended domains D_{nat} and D_{seq} are the obvious flat domains. Hence we have undefined or bottom objects denoted by `undefined-nat` and `undefined-seq`.

To formulate the specification we use the predicates `blegal` (for bounded legal) and `unsolvable`. The intended meaning of `(blegal a x b c m)` is that the sequence `a` is a defined answer to the bounded binpacking problem given by the blocks `x`= $i_0, \ldots, i_{p-1}$, the bins `b`= $j_0, \ldots, j_{q-1}$ and the bound `m`. This is to mean that `a` is a sequence `a`= $k_0, \ldots, k_{p-1}$, where k_n is the index of the bin the block i_n is to be packed into. Here the side condition that the first block i_0 is to be packed into a bin with index < `m` must be observed. So we require

$$\sum_{k_n=r} i_n \leq j_r \qquad \text{and} \qquad k_0 < m.$$

The answer `a=undefined-seq` is to mean that there is no solution.

The reason for the appearance of `c` is only technical: representing the bins by two sequences `b` and `c` allows to avoid the subtraction function. `b` remains fixed throughout, and `c` is initialized by a sequence of zeros of the same length as `b`.

The recursive definition of `blegal` is as follows.

```
blegal(k::a)(i::x)b c(m+1)
 =  blegal(k::a)(i::x)b c m
    v (k<=m & (ref c k)+i<=(ref b k) & blegal a x b(incr c k i)(len b))
blegal(k::a)(i::x)b c 0 = false
blegal nil(i::x)b c m   = false
blegal(k::a)nil b c m   = false
blegal nil nil b c m    = true
```

Here we have used some auxiliary functions: `(ref b k)` is the `k`–th member of the sequence `b`, and `(incr c k i)` is the sequence obtained from `c` by increasing its `k`–th element by `i`. Both functions should return `undefined-seq` if `k` is greater than or equal to the length `(len b)` (or `(len c)`).

To actually carry out proofs involving `blegal` it will be convenient to use the last four clauses in the definition of `blegal` as rewrite rules, and to encode the meaning of the essential first clause into two lemmata treated as "global assumptions" in the proof:

```
(add-global-assumption
 'lemma1 (pf "all a^,x,b,c,m.blegal a^ x b c m -> blegal a^ x b c(m+1)"))

(add-global-assumption
 'lemma2 (pf "all k,a^,i,x,b,c,m.k<=m
                                 -> (ref c k)+i<=(ref b k)
                                 -> blegal a^ x b(incr c k i)(len b)
                                 -> blegal(add k a^)(add i x)b c(m+1)"))
```

(`pf` means parse–formula here, and `(add i x)` is a prefix notation for `i::x`).

Now we could define unsolvable in terms of blegal by letting (unsolvable x b c m) denote all a.(blegal a x b c m) -> F, and then take as our specification or goal formula

```
(define goal-formula
 (pf "all x,m,b,c.(all a^.(delta-seq a^ -> (blegal a^ x b c m))
                          -> ((delta-seq a^ -> F) -> (unsolvable x b c m))
                          -> F)
                  -> F"))
```

i.e.

$$\forall x, m, b, c \exists \hat{a}.(\delta\hat{a} \to \texttt{blegal}\, \hat{a}xbcm) \land (\neg\delta\hat{a} \to \texttt{unsolvable}\, xbcm).$$

(The reason for using â instead of a here is to tell the machine that the value undefined-seq is allowed. (delta-seq â) means that â is a defined sequence.)

However, we run into a problem here. This goal formula is not of the form $\forall x \exists y\, A$ with quantifier–free A, since (unsolvable x b c m) by our definition contains a universal quantifier. Therefore we cannot guarantee that the general machinery for extracting programs from proofs will work here (cf. the example $\forall n \exists m \forall k. Tnnk \to Tnnm$ in Section 2).

Our solution to this problem is to observe that the inner universal is essentially bounded. Hence we can define unsolvable recursively, just as we have done for blegal. The intended meaning for (unsolvable x b c m) is that there is no possibility to pack the blocks x in such a way into the bins b-c that the first block is packed into a bin with index <m. The recursive definition of (unsolvable x b c m) is as follows.

```
unsolvable(i::x)b c(m+1)
 =  unsolvable(i::x)b c m
    & ((ref c m)+i<=(ref b m) -> unsolvable x b(incr c m i)(len b))
unsolvable(i::x)b c 0 = true
unsolvable nil b c m  = false
```

Now with this definition of unsolvable the goal formula has the right form $\forall x \exists y\, A$ with quantifier–free A.

To actually carry out proofs involving unsolvable it will again be convenient to use the last two clauses in the definition as rewrite rules, and to encode the meaning of the essential first clause into a lemma:

```
(add-global-assumption
 'lemma3 (pf "all i,x,b,c,m.
   unsolvable(add i x)b c m
   -> ((ref c m)+i<=(ref b m) -> unsolvable x b(incr c m i)(len b))
   -> unsolvable(add i x)b c(m+1)"))
```

In the proof of our goal we will make use of two more lemmata expressing properties of defined/undefined sequences.

```
(add-global-assumption
 'lemma4 (pf "all a^,m.delta-seq(add m a^) -> delta-seq a^"))

(add-global-assumption
 'lemma5 (pf "all a^,m.delta-seq a^ -> delta-seq(add m a^)"))
```

Let us now carry out the proof of our goal formula informally. The proof is by induction on the sequence `x`, and an auxiliary induction on `m`.

`i::x`, `m+1`. Choose â by hypothesis of the auxiliary induction.

Case `(delta-seq â)`. Take â and use Lemma 1.

Case `(delta-seq â) -> F`.

Subcase `(ref c m)+i<=(ref b m)`. Choose â1 by hypothesis of the (main) induction for the modified bins, i.e. for `x (len b) b (incr c m i)`. Take `m::â1`. To prove that this is a solution we need Lemmata 2 and 4 in the case `delta-seq m::â1`, and Lemmata 3 and 5 in the case `delta-seq m::â1 -> F`.

Subcase `(ref c m)+i<=(ref b m) -> F`. Unsolvable by Lemma 3.

`i::x`, `0`. Unsolvable.

`nil`, `m`. Take `nil`.

We have carried out this proof in the interactive prover MINLOG developed in Munich. In Appendix 1 the formal representation of this proof is shown. The two applications of induction and also the two uses of case distinctions can be seen clearly in the type–free λ–term representing the proof.

Note that the axiom scheme `if-at` taking care of case distinctions has the general form

$$(P(\vec{r}) \to A) \to (\neg P(\vec{r}) \to A) \to A,$$

where the atom $P(\vec{r})$ is given by a boolean term. Hence this axiom can be proved easily by boolean induction. In the particular case $A \equiv \bot$ — and only this case is used in our derivation — the axiom reads

$$(P(\vec{r}) \to \bot) \to ((P(\vec{r}) \to \bot) \to \bot) \to \bot,$$

and hence is easily derivable, even in minimal logic. However, we have kept it in the proof, since it helps to make the structure of the case distinctions in the proof more visible.

Let us now aim at demonstrating the pruning technique in this example. The first step consists in specializing the general problem to a situation where the length of the sequences of blocks and bins is fixed, say 2 for both, and `m` $=1$. Then we are to pack the blocks `i0::i1::nil` into the bins `j0::j1::nil` with the side condition that the first block `i0` has to go into the first bin `j0`. If we specialize the general proof to this particular situation and normalize it, then both inductions unfold completely and we obtain a proof as shown in Appendix 2.

Note that this proof has the following structure.

Case `u26`: $i_0 \leq j_0$.

Subcase `u27`: $i_0 + i_1 \leq j_0$. Take `0::0::nil`. To prove that this is a solution we need `u26`, `u27`, Lemmata 1 and 2 in the case `delta-seq 0::0::nil` *and Lemma 3 in the case* `delta-seq 0::0::nil -> F`.

Subcase `u31`: $i_0 + i_1 \not\leq j_0$.

Subsubcase `u32`: $i_1 \leq j_1$. Take `0::1::nil`. To prove that this is a solution we need `u26`, `u32` and Lemma 2 in the case `delta-seq 0::1::nil` *and* `u31` *and Lemmata 3 and 5 in the case* `delta-seq 0::1::nil -> F`.

Subsubcase `u38`: $i_1 \not\leq j_1$. Unsolvable by `u26`, `u31`, `u38` and Lemmata 2, 3 and 4.

Case `u44`: $i_0 \not\leq j_0$. Unsolvable by `u44` and Lemma 3.

Here for the first time we can observe that we have a proof produced by a machine and not by a human: It contains in its emphasized parts arguments a human would avoid to carry out, namely to start from some obviously false assumption (`u29:delta-seq 0::0::nil -> F` in the first instance and `u34:delta-seq 0::1::nil -> F` in the second instance) and then carry out a somewhat elaborate argument to arrive at some conclusion (as it turns out, atomic formulas in both instances). We could use the first pruning step to remove these and other unneccessary parts of the proof. However, such an application of pruning does not yet show the effect we want to demonstrate, since the algorithm represented by the proof has not changed.

To achieve this desired effect, we first have to provide some additional assumptions on the input data. So let us assume that the second block is less than or equal to the first one, and that the first bin is less than or equal to the second one, i.e. that

$$\texttt{v1}\colon i_1 \leq i_0 \qquad \text{and} \qquad \texttt{v2}\colon j_0 \leq j_1$$

holds. Then we can replace quite a number of subproofs by different ones — this is the first pruning step of Section 2 —, where we always aim at replacing the use of assumption variables bound later in the proof (e.g. case assumptions) by the use of our additional assumptions `v1` and `v2` and — if necessary — some other assumption variables. Let us look at these replacements individually. The result is shown in Appendix 3.

(1) Here the machine replaced the derivation using Lemmata 3 and 5 of an atom from the false assumption `u29:delta-seq 0::0::nil -> F` — which rewrites into `T -> F` — by a trivial derivation using ex–falso–quodlibet and the truth–axiom `T`.

(2) The case assumption `u32`: $i_1 \leq j_1$ is replaced by a simple derivation (involving a global assumption `trans` expressing transitivity of $\leq$) of $i_1 \leq j_1$ from the additional assumptions `v1`: $i_1 \leq i_0$ and `v2`: $j_0 \leq j_1$, together with the outer case assumption `u26`: $i_0 \leq j_0$.

(3) Similar to (1): The lengthy derivation using Lemmata 3 and 5 of an atom from the false assumption `u34:delta-seq 0::1::nil -> F` — which rewrites into `T -> F` — is replaced by a trivial derivation using ex–falso–quodlibet and the truth–axiom `T`. Note that an occurrence of the case assumption `u31`: $i_0 + i_1 \not\leq j_0$ present in the original proof has disappeared.

(4) Here the derivation `((lemma4 undefined-seq) 0) u39` of $\bot$ has been replaced by `u39:`$\bot$. We have used here that `delta-seq undefined-seq` rewrites into $\bot$ and `0::undefined-seq` rewrites into `undefined-seq`,

(5) Here the subproof `(lambda (u43) ((efq false) (u38 u43)))` of $i_1 \not\leq j_1$ is replaced by `u38`, which derives the same formula.

(6) As in (5), we could replace the subproof `(lambda (u42) ((efq false) (u31 u42)))` of $i_0 + i_1 \not\leq j_0$ by `u31:`$i_0 + i_1 \not\leq j_0$. However, here we can go one step further and replace the subproof by a simple derivation (involving a global assumption `<=-+-ax` expressing compatibility of $\leq$ with $+$ and again `trans` expressing transitivity of $\leq$) of $i_0 + i_1 \not\leq j_0$ from the additional assumptions `v1:`$i_1 \leq i_0$ and `v2:`$j_0 \leq j_1$, together with the outer case assumption `u38:`$i_1 \not\leq j_1$.

(7) Similar to (5).

Now we come to the essential second pruning step. It consists in going through every node of the derivation (from the bottom upwards) and checking whether the formula A at this node also appears somewhere further up in the proof in such a way that between the two occurrences of A no $\rightarrow^+$–rule appears which binds an assumption variable that actually occurs free further up in the proof tree.

In the subcase `u31:`$i_0 + i_1 \not\leq j_0$ all occurrences of the case assumption `u31` have been pruned away. Hence the proof of $\bot$ provided by this subderivation is independent of `u31`. We can remove the case axiom and replace the subderivation of $\bot$ starting with `((if-at 'false)` $i_0 + i_1 \leq j_0$`)` by the proof of that case starting with `((if-at 'false)` $i_1 \leq j_1$`)`.

Moreover, the proof of $\bot$ starting with this case axiom `((if-at 'false)` $i_1 \leq j_1$`)` has again the property that its first subsubcase `u32:`$i_1 \leq j_1$ does not use its case assumption `u32`: it has been pruned away in step (2). So again we can remove the case axiom and replace the subderivation of $\bot$ starting with `((if-at 'false)` $i_1 \leq j_1$`)` by the proof of that case. The resulting derivation is shown in Appendix 4.

Note that we have indeed changed the program provided by our classical proof in an extensional way. In the general situation — without additional assumptions — our proof yielded `0::0::nil` as an answer for the particular arguments i_0, i_1 and j_0, j_1 satisfying $i_0 + i_1 \leq j_0$, $i_1 \leq i_0$ and $j_0 \leq j_1$ (see Appendix 2). After the pruning procedure, the resulting proof yields `0::1::nil` as an answer for these particular arguments satisfying the same conditions (see Appendix 4).

Appendix 1

Proof of the general goal formula for the one–dimensional binpacking problem.

```
#(() (all x (all m (all b (all c (imp (all a^ (imp (imp (atom
(delta-seq a^)) (atom (((((blegal a^) x) b) c) m))) (imp (imp (imp
(atom (delta-seq a^)) (atom false)) (atom ((((unsolvable x) b) c) m)))
(atom false)))) (atom false)))))) (lambda (x) ((((seq-ind-at '(all x
(all m (all b (all c (imp (all a^ (imp (imp (atom (delta-seq a^))
(atom (((((blegal a^) x) b) c) m))) (imp (imp (imp (atom (delta-seq
a^)) (atom false)) (atom ((((unsolvable x) b) c) m))) (atom false))))
(atom false))))))) (lambda (m) (lambda (b) (lambda (c) (lambda (u22)
(((u22 nil) (lambda (u23) truth-axiom-symbol)) (lambda (u24) ((efq
((((unsolvable nil) b) c) m)) (u24 truth-axiom-symbol))))))))) (lambda
(x) (lambda (n10) (lambda (u11) ((((lambda (x) (lambda (i) (lambda
(IV) (((((natind-at '(all m (all b (all c (imp (all a^ (imp (imp (atom
(delta-seq a^)) (atom (((((blegal a^) ((add i) x)) b) c) m))) (imp
(imp (imp (atom (delta-seq a^)) (atom false)) (atom ((((unsolvable
((add i) x)) b) c) m))) (atom false)))) (atom false)))))) i) x)
(lambda (b) (lambda (c) (lambda (u19) (((u19 undefined-seq) (lambda
(u20) ((efq (((((blegal undefined-seq) ((add i) x)) b) c) 0)) u20)))
(lambda (u21) truth-axiom-symbol)))))) (lambda (m) (lambda (NIV)
(lambda (b) (lambda (c) (lambda (u1) (((NIV b) c) (lambda (a^) (lambda
(u2) (lambda (u3) ((((if-at '(atom false)) (delta-seq a^)) (lambda
(u4) (((u1 a^) (lambda (u5) ((((((lemma1 a^) ((add i) x)) b) c) m) (u2
u4)))) (lambda (u6) ((efq ((((unsolvable ((add i) x)) b) c) ((+-nat m)
1))) (u6 u4)))))) (lambda (u7) ((((if-at '(atom false)) (<=-nat
((+-nat ((ref c) m)) i) ((ref b) m))) (lambda (u8) ((((IV (len b)) b)
(((incr c) m) i)) (lambda (a^1) (lambda (u9) (lambda (u10) (((u1 ((add
m) a^1)) (lambda (u11) (((((((((lemma2 m) a^1) i) x) b) c) m)
truth-axiom-symbol) u8) (u9 (((lemma4 a^1) m) u11))))) (lambda (u12)
(((((((lemma3 i) x) b) c) m) (u3 u7)) (lambda (u13) (u10 (lambda (u14)
(u12 (((lemma5 a^1) m) u14))))))))))))) (lambda (u15) (((u1 a^)
(lambda (u16) ((efq (((((blegal a^) ((add i) x)) b) c) ((+-nat m) 1)))
(u7 u16)))) (lambda (u17) (((((((lemma3 i) x) b) c) m) (u3 u17))
(lambda (u18) ((efq ((((unsolvable x) b) (((incr c) m) i)) (len b)))
(u15 u18)))))))))))))))))))))))) x) n10) u11))))) x)))
```

Appendix 2

This is the result of specializing the general problem to a situation where the length of the sequences of blocks and bins is fixed, say 2 for both, and `m` = 1. We are to pack the blocks `i0::i1::nil` into the bins `j0::j1::nil` with the side condition that the first block `i0` has to go into the first bin `j0`. If we specialize the general proof to this particular situation and normalize it, then both inductions unfold completely and we obtain the following type–free proof term.

```
(lambda (u25)
 ((((if-at '(atom false)) (<=-nat i0 j0))
   (lambda (u26)
    ((((if-at '(atom false)) (<=-nat ((+-nat i0) i1) j0))
      (lambda (u27)
       (((u25 ((add 0) ((add 0) nil)))
         (lambda (u28)
          (((((((((lemma2 0) ((add 0) nil)) i0) ((add i1) nil))
                ((add j0) ((add j1) nil)))
               ((add 0) ((add 0) nil)))
              0)
             truth-axiom-symbol)
            u26)
           ((((((lemma1 ((add 0) nil)) ((add i1) nil))
               ((add j0) ((add j1) nil)))
              ((add i0) ((add 0) nil)))
             1)
            (((((((((lemma2 0) nil) i1) nil)
                    ((add j0) ((add j1) nil)))
                 ((add i0) ((add 0) nil)))
                0)
               truth-axiom-symbol)
              u27)
             truth-axiom-symbol)))))
        (lambda (u29)
(1)      ((((((lemma3 i0) ((add i1) nil)) ((add j0) ((add j1) nil)))
 |           ((add 0) ((add 0) nil)))
 |          0)
 |         truth-axiom-symbol)
 |        (lambda (u30)
 |         ((efq false)
 |          (u29 (((lemma5 ((add 0) nil)) 0)
(1)               truth-axiom-symbol)))))))))
     (lambda (u31)
```

```
      ((((if-at '(atom false)) (<=-nat i1 j1))
        (lambda (u32)
         (((u25 ((add 0) ((add 1) nil)))
           (lambda (u33)
            (((((((((lemma2 0) ((add 1) nil)) i0) ((add i1) nil))
                  ((add j0) ((add j1) nil)))
                 ((add 0) ((add 0) nil)))
                0)
               truth-axiom-symbol)
              u26)
             (((((((((lemma2 1) nil) i1) nil)
                    ((add j0) ((add j1) nil)))
                  ((add i0) ((add 0) nil)))
                 1)
                truth-axiom-symbol)
(2)            u32)
              truth-axiom-symbol))))
          (lambda (u34)
(3)        (((((((lemma3 i0) ((add i1) nil))
 |              ((add j0) ((add j1) nil)))
 |             ((add 0) ((add 0) nil)))
 |            0)
 |           truth-axiom-symbol)
 |          (lambda (u35)
 |           (((((((lemma3 i1) nil) ((add j0) ((add j1) nil)))
 |               ((add i0) ((add 0) nil)))
 |              1)
 |             (((((((lemma3 i1) nil) ((add j0) ((add j1) nil)))
 |                 ((add i0) ((add 0) nil)))
 |                0)
 |               truth-axiom-symbol)
 |              (lambda (u36) ((efq false) (u31 u36)))))
 |            (lambda (u37)
 |             ((efq false)
 |              (u34
 |               (((lemma5 ((add 1) nil)) 0)
(3)             (((lemma5 nil) 1) truth-axiom-symbol)))))))))))
       (lambda (u38)
        (((u25 undefined-seq)
          (lambda (u39)
           (((((((((lemma2 0) undefined-seq) i0) ((add i1) nil))
                 ((add j0) ((add j1) nil)))
                ((add 0) ((add 0) nil)))
               0)
```

```
              truth-axiom-symbol)
             u26)
            ((efq
              (((((blegal undefined-seq) ((add i1) nil))
                 ((add j0) ((add j1) nil)))
                ((add i0) ((add 0) nil)))
               2))
(4)          (((lemma4 undefined-seq) 0) u39)))))
         (lambda (u40)
          ((((((lemma3 i0) ((add i1) nil))
               ((add j0) ((add j1) nil)))
              ((add 0) ((add 0) nil)))
             0)
            truth-axiom-symbol)
           (lambda (u41)
            (((((((lemma3 i1) nil) ((add j0) ((add j1) nil)))
                ((add i0) ((add 0) nil)))
               1)
              (((((((lemma3 i1) nil) ((add j0) ((add j1) nil)))
                  ((add i0) ((add 0) nil)))
                 0)
                truth-axiom-symbol)
(6)            (lambda (u42) ((efq false) (u31 u42)))))
(5)          (lambda (u43) ((efq false) (u38 u43))))))))))))
  (lambda (u44)
   (((u25 undefined-seq)
     (efq
      (((((blegal undefined-seq) ((add i0) ((add i1) nil)))
         ((add j0) ((add j1) nil)))
        ((add 0) ((add 0) nil)))
       1)))
    (lambda (u46)
     (((((((lemma3 i0) ((add i1) nil)) ((add j0) ((add j1) nil)))
         ((add 0) ((add 0) nil)))
        0)
       truth-axiom-symbol)
(7)   (lambda (u47) ((efq false) (u44 u47))))))))
```

Appendix 3

Here we assume that we have the additional assumptions

$$\mathtt{v1}\colon i_1 \leq i_0 \qquad \text{and} \qquad \mathtt{v2}\colon j_0 \leq j_1.$$

Then we can replace quite a number of subproofs by different ones — this is the first pruning step of Section 2 —, where we always aim at replacing the use of assumption variables bound later in the proof (e.g. case assumptions) by the use of our additional assumptions `v1` and `v2` and — if necessary — some other assumption variables. The resulting proof term looks as follows.

```
(lambda (u25)
 ((((if-at '(atom false)) (<=-nat i0 j0))
   (lambda (u26)
    ((((if-at '(atom false)) (<=-nat ((+-nat i0) i1) j0))
      (lambda (u27)
       (((u25 ((add 0) ((add 0) nil)))
         (lambda (u28)
          (((((((((lemma2 0) ((add 0) nil)) i0) ((add i1) nil))
                ((add j0) ((add j1) nil)))
               ((add 0) ((add 0) nil)))
              0)
             truth-axiom-symbol)
            u26)
           ((((((lemma1 ((add 0) nil)) ((add i1) nil))
               ((add j0) ((add j1) nil)))
              ((add i0) ((add 0) nil)))
             1)
            ((((((((((lemma2 0) nil) i1) nil) ((add j0) ((add j1) nil)))
                 ((add i0) ((add 0) nil)))
                0)
               truth-axiom-symbol)
              u27)
             truth-axiom-symbol)))))
        (lambda (u29)
(1)      ((efq
 |         ((and-strict true)
 |          ((imp-strict ((leq-strict-nat i0) j1)) false)))
(1)       (u29 truth-axiom-symbol))))))
     (lambda (u31)
      ((((if-at '(atom false)) (<=-nat i1 j1))
        (lambda (u32)
         (((u25 ((add 0) ((add 1) nil)))
```

```
           (lambda (u33)
            (((((((((lemma2 0) ((add 1) nil)) i0) ((add i1) nil))
                  ((add j0) ((add j1) nil)))
                 ((add 0) ((add 0) nil)))
                0)
               truth-axiom-symbol)
              u26)
             (((((((((lemma2 1) nil) i1) nil) ((add j0) ((add j1) nil)))
                  ((add i0) ((add 0) nil)))
                 1)
                truth-axiom-symbol)
(2)            (((((trans i1) j0) j1)
 |               (((((trans i1) i0) j0) v1) u26)) v2))
(2)           truth-axiom-symbol))))
          (lambda (u34)
(3)        ((efq
 |           ((and-strict true)
 |            ((imp-strict ((leq-strict-nat i0) j1)) false)))
(3)         (u34 truth-axiom-symbol))))))
       (lambda (u38)
        (((u25 undefined-seq)
          (lambda (u39)
           (((((((((lemma2 0) undefined-seq) i0) ((add i1) nil))
                 ((add j0) ((add j1) nil)))
                ((add 0) ((add 0) nil)))
               0)
              truth-axiom-symbol)
             u26)
            ((efq
              (((((blegal undefined-seq) ((add i1) nil))
                 ((add j0) ((add j1) nil)))
                ((add i0) ((add 0) nil)))
               2))
(4)          u39))))
         (lambda (u40)
          (((((((lemma3 i0) ((add i1) nil)) ((add j0) ((add j1) nil)))
              ((add 0) ((add 0) nil)))
             0)
            truth-axiom-symbol)
           (lambda (u41)
            (((((((lemma3 i1) nil) ((add j0) ((add j1) nil)))
                ((add i0) ((add 0) nil)))
               1)
              ((((((lemma3 i1) nil) ((add j0) ((add j1) nil)))
```

```
                  ((add i0) ((add 0) nil)))
                 0)
                truth-axiom-symbol)
(6)            (lambda (u48)
 |              (u38
 |               (((((trans i1) j0) j1)
 |                 (((((trans i1) i0) j0) v1)
 |                  ((((<=-+-ax i0) i1) j0) u48)))
(6)               v2)))))
             u38)))))))))))
  (lambda (u44)
   (((u25 undefined-seq)
     (efq
      (((((blegal undefined-seq) ((add i0) ((add i1) nil)))
         ((add j0) ((add j1) nil)))
        ((add 0) ((add 0) nil)))
       1)))
    (lambda (u46)
     (((((((lemma3 i0) ((add i1) nil)) ((add j0) ((add j1) nil)))
         ((add 0) ((add 0) nil)))
        0)
       truth-axiom-symbol)
(7)   u44))))))
```

Appendix 4

This is the result of applying the second pruning step to the derivation shown in Appendix 3.

```
(lambda (u25)
 ((((if-at '(atom false)) (<=-nat i0 j0))
   (lambda (u26)
    (((u25 ((add 0) ((add 1) nil)))
      (lambda (u33)
       ((((((((((lemma2 0) ((add 1) nil)) i0) ((add i1) nil))
             ((add j0) ((add j1) nil)))
            ((add 0) ((add 0) nil)))
           0)
          truth-axiom-symbol)
         u26)
        (((((((((lemma2 1) nil) i1) nil) ((add j0) ((add j1) nil)))
             ((add i0) ((add 0) nil)))
            1)
           truth-axiom-symbol)
          (((((trans i1) j0) j1) (((((trans i1) i0) j0) v1) u26)) v2))
         truth-axiom-symbol))))
     (lambda (u34)
      ((efq ((and-strict true)
             ((imp-strict ((leq-strict-nat i0) j1)) false)))
       (u34 truth-axiom-symbol))))))
  (lambda (u44)
   (((u25 undefined-seq)
     (efq
      (((((blegal undefined-seq) ((add i0) ((add i1) nil)))
         ((add j0) ((add j1) nil)))
        ((add 0) ((add 0) nil)))
       1)))
    (lambda (u46)
     (((((((lemma3 i0) ((add i1) nil)) ((add j0) ((add j1) nil)))
         ((add 0) ((add 0) nil)))
        0)
       truth-axiom-symbol)
      u44))))))
```

References

[1] Ulrich Berger. Program extraction from normalization proofs. In M. Bezem and J.F. Groote, editors, *Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 91–106, Berlin, 1993. Springer.

[2] Roy Dyckhoff. Contraction–free sequent calculi for intuitionistic logic. *Journal of Symbolic Logic*, 57:793–807, 1992.

[3] Harvey Friedman. Classically and intuitionistically provably recursive functions. In Dana S. Scott and Gert H. Müller, editors, *Higher Set Theory*, pages 21–28. Springer Lecture Notes in Mathematics, Volume 669, 1978.

[4] Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1934.

[5] Christopher Alan Goad. *Computational uses of the manipulation of formal proofs.* PhD thesis, Stanford University, August 1980. Stanford Department of Computer Science Report No. STAN–CS–80–819.

[6] Jörg Hudelmaier. Bounds for cut elimination in intuitionistic propositional logic. *Archive for Mathematical Logic*, 31:331–354, 1992.

[7] Dale Miller. A logic programming language with lambda–abstraction, function variables and simple unification. *Journal of Logic and Computation*, 2(4):497–536, 1991.

[8] G.E. Mints. On e–theorems (in russian). *Zapiski*, 40:110–118, 1974.

[9] Chetan Murthy. Extracting constructive content from classical proofs. Technical Report 90–1151, Dep.of Comp.Science, Cornell Univ., Ithaca, New York, 1990. PhD thesis.

[10] Tobias Nipkow. Functional unification of higher–order patterns. In *Proc. 8th IEEE Symp. Logic in Computer Science*, pages 64–74, 1993.

[11] Dag Prawitz. *Natural Deduction*, volume 3 of *Acta Universitatis Stockholmiensis. Stockholm Studies in Philosophy.* Almqvist & Wiksell, Stockholm, 1965.

[12] Dan Sahlin, Torkel Franzen, and Seif Haridi. An intuitionistic predicate logic theorem prover. *Journal of Logic and Computation*, 2(6):619–656, 1992.

[13] Helmut Schwichtenberg. Proofs as programs. In P. Aczel, H. Simmons, and S.S. Wainer, editors, *Proof Theory. A selection of papers from the Leeds Proof Theory Programme 1990*, pages 81–113. Cambridge University Press, 1992.

[14] N. Shankar. Proof search in intuitionistic sequent calculus, 1991.

[15] Wilfried Sieg and Richard Scheines. Searching for proofs (in sentential logic). In Leslie Burkholder, editor, *Philosophy and the computer*, pages 137–159, Boulder, San Francisco, Oxford, 1992. Westview Press.

[16] Martin Stein. *Interpretationen der Heyting–Arithmetik endlicher Typen.* PhD thesis, Universität Münster, Fachbereich Mathematik, 1976.

[17] Anne S. Troelstra, editor. *Metamathematical Investigations of Intuitionistic Arithmetic and Analysis*, volume 344 of *Lecture Notes in Mathematics.* Springer, Berlin, 1973.

[18] Anne S. Troelstra and Dirk van Dalen. *Constructivism in Mathematics. An Introduction*, volume 121, 123 of *Studies in Logic and the Foundations of Mathematics.* North–Holland, Amsterdam, 1988.

[19] Anton Wallner. Komplexe Existenzbeweise in der Arithmetik. Master's thesis, Mathematisches Institut der Universität München, 1993.

[20] Hermann Weyl. Über die neue Grundlagenkrise der Mathematik. *Mathematische Zeitschrift*, 10, 1921.