Program Extraction

Monika Seisenberger

Swansea University

Proof Society Summer School, Ghent, 2–5 September 2018

Outline of the tutorial

Part 1 Extraction of Programs from Constructive Proofs:

- 1.1 Introduction, Curry Howard-Correspondence, Realizability Interpretation
- 1.2 Tool support, Examples, Extension to inductive Definitions

Part 2 Extraction of Programs from Classical Proofs

- 2.1 A-translation, Choice principles
- 2.2 Applications for both parts

Intuition of the connection between a proof and a program

Brouwer-Heyting-Kolmogorov Interpretation:
assigns a constructive meaning to each logical connective and quantifier.

- A proof of a conjunction $A \wedge B$ is given by a pair (p_1, p_2) such that p_1 proves A and p_2 proves B.
- A proof of a disjunction $A \vee B$ is given by (i, p) where either $i = 0$ and p proves A or $i = 1$ and p proves B.
- A proof an existence statement $\exists x A$ is given by a witness a and a proof that $A(a)$ holds.
- A proof of an universal statement $\forall x A$ is given by a "construction or method" transforming an arbitrary individual a into a proof a $A(a)$
- A proof of an implication $A \to B$ is given by a construction or method that transforms a proof of A into a proof of B.

$\neg A$ is treated like $A \to \bot$. There is no proof of $\bot$. For prime formulas the notion of proof is supposed to be given. Note there is no explanation of what amounts to a "construction or method."

A first example

Example (Quotient and Remainder)

$$\forall a, b.\ 0 < b \to \exists q, r.\ a = q * b + r \wedge r < b.$$

Proof: Ind(a). Base $a = 0$: take $q = 0$, $r = 0$.
Step: Given q, r such that $a = q * b + r$ and $r < b$, find new q', r' for $a + 1$

A first example

Example (Quotient and Remainder)

$$\forall a, b.\ 0 < b \rightarrow \exists q, r.\ a = q * b + r \wedge r < b.$$

Proof: Ind(a). Base $a = 0$: take $q = 0$, $r = 0$.
Step: Given q, r such that $a = q * b + r$ and $r < b$, find new q', r' for $a + 1$

Program extraction will yield a program with solves the computational problem:

> Input: two numbers a,b
> Output: a pair of numbers (q,r) with the desired property.

In the following, we will demonstrate what we need on the formal side to get to this program. We first explain the logical system, in particular how to store proofs as proof terms, then we show how to extract the provably correct program.

Methods to make the constructive content explicit

There are various methods to make the constructive content of a proof explicit

- Cut elimination
- Realizability (Kleene, Kreisel),
- Dialectica interpretation (Goedel)
- Proofs as Programs in Type Theory
- Classical Realizability (Krivine)

In this tutorial we focus on realizability as it is the most direct technique. The technique will be demonstrated in the interactive theorem prover Minlog, which can extract programs from both constructive and classical proofs. Other theorem provers like Coq or Agda could also be used for the constructive examples. [Note Adga would lead to dependently typed programs.]

Logic Background

Our proof calculus is HA$^\mu$ which is an extension of Heyting Arithmetic in finite types, HA$^\omega$ (see Troelstra'73 or Troelstra/vanDalen'88) by inductively defined types.
Cf also: Theory of Computable Functionals (in Schwichtenberg/Wainer, Proofs and Computations, Perspectives in Logic,'12)

Definition (Types)

Types are generated from inductive types, via $\times$ and $\to$, that is, if ρ and σ are types, then so are $\rho \times \sigma$ and $\rho \to \sigma$; in short: types are

$$\mu |\, \rho \times \sigma \,|\, \rho \to \sigma.$$

Note for our first example we only need the two inductive types Bool = True + False, and Nat = Zero + Succ(Nat). [You may skip the next slide.]

Extended Heyting Arithmetic (cont.)

Inductive types in their most general form look as follows. [That means the user can define further inductive types according to this construction, and inductive types of this form will later also be automatically generated as realizers for inductive definitions.]

Definition (Inductive Type)

A new inductive type μ is introduced by the following equation:

$$\begin{aligned}\mu \quad = \quad & c_1\big(\vec{\rho_1},\ \vec{\sigma_{11}} \to \mu,\ \ldots,\ \vec{\sigma_{1m_1}} \to \mu\big) \\ & + \cdots + \\ & c_n\big(\vec{\rho_n},\ \vec{\sigma_{n1}} \to \mu,\ \ldots,\ \vec{\sigma_{nm_n}} \to \mu\big)\end{aligned}$$

where for all $i < n, j < m_i$ such that $0 \leq m_i, n$, $\vec{\rho_i}$ and $\vec{\sigma_{ij}}$ are lists of types built from previously defined types only. Then, μ is the type whose elements are generated from the constructors

$$c_i : \vec{\rho_i} \to \overrightarrow{\vec{\sigma_i} \to \mu} \to \mu.$$

Extended Heyting Arithmetic (cont.)

Definition (Terms)

are built from typed variables and constants via λ-abstraction, application, pairing and projection, that is, terms are

$$x \mid c \mid \lambda x t \mid st \mid \langle s, t\rangle \mid \pi_i(t).$$

For each ground type μ and type τ we have a recursion operator $R_{\mu,\tau}$. If $\vec{\rho}_i \to \overrightarrow{\vec{\sigma}_i \to \mu} \to \mu$ is the type of the i-th constructor c_i of μ, then the i-th step type δ_i is $\vec{\rho}_i \to \overrightarrow{\vec{\sigma}_i \to \mu} \to \overrightarrow{\vec{\sigma}_i \to \tau} \to \tau$ and the recursion operator has the type

$$\mathrm{R}_{\mu,\tau} : \delta_1 \to \cdots \to \delta_n \to \mu \to \tau.$$

Analogously, we also have a case distinction operator

$$C_{\mu,\tau} : \delta_1 \to \cdots \to \delta_n \to \mu \to \tau$$

where the i-th step type δ_i simplifies to $\vec{\rho}_i \to \overrightarrow{\vec{\sigma}_i \to \mu} \to \tau$.

Conversions

The conversion rules are

$$\begin{array}{rcl} (\lambda x t)s & \mapsto & t[x/s] \\ (\lambda x t)x & \mapsto & t, \quad x \notin \mathrm{FV}(t) \\ \pi_i(\langle t_0, t_1 \rangle) & \mapsto & t_i,\ i = 0, 1 \\ \langle \pi_0(t), \pi_1(t) \rangle & \mapsto & t \end{array}$$

With regard to the recursion operator, assuming that $\vec{t}$ consists of parameter arguments ${t_1}^P, \ldots, {t_m}^P$ and recursive arguments ${t_1}^R, \ldots, {t_n}^R$, we have the conversion rule

$$\mathrm{R}_{\mu,\tau}\vec{s}(c_i\vec{t}) \mapsto s_i\vec{t}(\mathrm{R}_{\mu,\tau}\vec{s} \circ {t_1}^R) \ldots (\mathrm{R}_{\mu,\tau}\vec{s} \circ {t_n}^R)$$

where $r^{\sigma \to \tau} \circ t^{\vec{\rho} \to \sigma} := \lambda \vec{y}^{\vec{\rho}}.(r(t\vec{y}))$.

The analogous rule for the case distinction operator is

$$C_{\mu,\tau}\vec{s}(c_i\vec{t}) \mapsto s_i\vec{t}.$$

Formulas

Definition (Formulas)

Let $\mathcal{P}$ be a set of predicate symbols, each of a fixed arity $\vec{\rho} = \rho_1, \ldots, \rho_n$. We always assume $\mathcal{P}$ to contain a nullary predicate symbol $\perp$ and a predicate symbol atom of arity boole. Formulas are built from atomic formulas $P(\vec{t})$ ($P \in \mathcal{P}$) via implication, conjunction and quantification. Hence formulas are

$$P(\vec{t}) \mid A \to B \mid A \wedge B \mid \forall x^\rho A \mid \exists x^\rho A.$$

where in $P(\vec{t})$ we assume that P is of arity $\vec{\rho}$ and the terms $\vec{t} = t_1, \ldots, t_n$ are of types $\rho_1, \ldots, \rho_n$ respectively.

Definition (Type of a Formula)

is either a type or the symbol $*$ (for "not computationally meaningful").

$$\tau(P(\vec{t})) := \begin{cases} \tau_0(P) & \text{if } P \in \mathcal{P} \text{ with assigned } \tau_0(P) \\ * & \text{otherwise.} \end{cases}$$

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

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

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

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

Natural Deduction

Definition (Proofs)

Proofs are presented as lambda terms via the Curry-Howard Correspondence.

$$
\begin{aligned}
& u^A \mid c^A \ (c \text{ an axiom}) \\
& \mid (\lambda u^A d^B)^{A \to B} \mid (d^{A \to B} e^A)^B \mid \\
& (\langle d^A, e^B \rangle)^{A \wedge B} \mid (\pi_0(d^{A \wedge B}))^A \mid \pi_1((d^{A \wedge B}))^B \mid \\
& (d^{\forall x A} t)^{A(t)} \mid (\lambda x d)^{\forall x A}, \ x \notin \mathrm{FV}(C) \text{ for } u^C \in \mathrm{FA}(d)
\end{aligned}
$$

where for a given derivation d, $\mathrm{FA}(d)$ is the set of free assumptions in d.

Axioms

1) For existential quantifier:

$$
\begin{array}{ll}
(\exists^{+}_{x,A}) & \forall x.A \to \exists x A \\
(\exists^{-}_{x,A,B}) & \exists x A \to (\forall x.A \to B) \to B, \quad (x \notin \mathrm{FV}(B))
\end{array}
$$

2) Logical axioms Truth: T and efq-axioms $\bot \to A$ for any formula A.
3) Axioms for equality, such as reflexivity, transitivity, symmetry and compatibility, e.g.,

$$(\text{Compat}) \quad \forall \vec{x}, x, y.x = y \to P(x) \to P(y).$$

4) For each algebra, μ, we have axioms for case distinction and induction, denoted by $\text{Cases}_{\mu,A}$ and $\text{Ind}_{\mu,A}$.

First, the existential quantifier are treated by the axioms:

$$
\begin{array}{lll}
(\exists^+_{x,A}) & \forall x.A \to \exists x A & \\
(\exists^-_{x,A,B}) & \exists x A \to (\forall x.A \to B) \to B, & (x \notin \mathrm{FV}(B)) \\
(\exists^{\mathsf{nc}+}_{x,A}) & \forall^{\mathsf{nc}} x.A \to \exists^{\mathsf{nc}} x A & \\
(\exists^{\mathsf{nc}-}_{x,A,B}) & \exists^{\mathsf{nc}} x A \to (\forall^{\mathsf{nc}} x.A \to B) \to B, & (x \notin \mathrm{FV}(B))
\end{array}
$$

Realizability: Definition (Extracted Program)

Given a derivation d of a computationally meaningful formula A, we inductively define the extracted term ("program") $[\![d]\!]$ of type $\tau(A)$.

$$
\begin{aligned}
[\![u^A]\!] &:= x_A^{\tau(A)} \text{(preassigned)} \\
[\![\lambda u^A d]\!] &:= \begin{cases} [\![d]\!] & \text{if } \tau(A) = *, \\ \lambda x_u^{\tau(A)} [\![d]\!] & \text{otherwise.} \end{cases} \\
[\![d^{A \to B} e]\!] &:= \begin{cases} [\![d]\!] & \text{if } \tau(A) = *, \\ [\![d]\!][\![e]\!] & \text{otherwise.} \end{cases} \\
[\![\langle d_0^{A_0}, d_1^{A_1} \rangle]\!] &:= \begin{cases} [\![d_i]\!] & \text{if } \tau(A_{1-i}) = * \\ \langle [\![d_0]\!], [\![d_1]\!] \rangle & \text{otherwise.} \end{cases} \\
[\![\pi_i(d^{A_0 \wedge A_1})]\!] &:= \begin{cases} [\![d]\!] & \text{if } \tau(A_{1-i}) = * \\ \pi_i [\![d]\!] & \text{otherwise.} \end{cases} \\
[\![(\lambda x d)^{\forall x A}]\!] &:= \lambda x [\![d]\!], \\
[\![d^{\forall x A} t]\!] &:= [\![d]\!] t.
\end{aligned}
$$

Extracted terms (realizers) for axioms

1) The extracted terms for the $\exists$-axioms are

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

$$[\![\exists^-_{x^\rho,A,B}]\!] := \begin{cases} \lambda x^\rho \lambda f^{\rho\to\tau(B)} fx & \text{if } \tau(A) = *, \\ \lambda z^{\rho\times\tau(A)} \lambda f^{\rho\to\tau(A)\to\tau(B)} f\pi_0(z)\pi_1(z) & \text{otherwise.} \end{cases}$$

2) For a formula A with $\tau(A) \neq *$, the efq axioms $F \to A$ and $\bot \to A$ are realized by a canonical inhabitant of the type $\tau(A)$.
3) The compatibility axiom, $\forall^{\mathrm{nc}}\vec{x}, x, y.x = y \to A(x) \to A(y)$, is realized by $\lambda x^{\tau(A)} x$.
4) Induction and case distinction on an inductive datatype μ, $\mathrm{Ind}_{\mu,A}$ and $\mathrm{Cases}_{\mu,A}$, correspond to recursion on this datatype, $\mathrm{R}_{\mu,\tau(A)}$, case distinction, $C_{\mu,\tau(A)}$, respectively.

Modified Realizability (Kreisel)

For every formula A we define a formula r **mr** A where r is either a term of type $\tau(A)$ or the symbol ϵ depending on whether or not A is computationally meaningful.

We assume that for each predicate $P : \rho_1, \dots, \rho_n$ with computational content we have enriched our language by a predicate $\tilde{P}$ of arity $\tau_0(P), \rho_1, \dots, \rho_n$.

$$r \ \mathbf{mr} \ P(\vec{t}) \quad := \quad \begin{cases} \tilde{P}(r, \vec{t}) & \text{if } P(\vec{t}) \text{ has computational content} \\ P(\vec{t}) & \text{otherwise} \end{cases}$$

[For our first example, we do no have any atomic formulas/predicates with computational content.]

Modified Realizability (cont)

$$r \ \mathbf{mr}\ (A \to B) \quad := \quad \begin{cases} \epsilon \ \mathbf{mr}\ A \to r \ \mathbf{mr}\ B & \text{if } \tau(A) = *, \\ \forall x.x \ \mathbf{mr}\ A \to \epsilon \ \mathbf{mr}\ B & \text{if } \tau(A) \neq * = \tau(B), \\ \forall x.x \ \mathbf{mr}\ A \to rx \ \mathbf{mr}\ B & \text{otherwise.} \end{cases}$$

$$r \ \mathbf{mr}\ (A_0 \wedge A_1) \quad := \quad \begin{cases} r \ \mathbf{mr}\ A_{1-i} \wedge \ \epsilon \ \mathbf{mr}\ A_i & \text{if } \tau(A_i) = *, \\ \pi_0(r) \ \mathbf{mr}\ A_0 \wedge \pi_1(r) \ \mathbf{mr}\ A_1 & \text{otherwise.} \end{cases}$$

$$r \ \mathbf{mr}\ \forall x A \quad := \quad \begin{cases} \forall x.\epsilon \ \mathbf{mr}\ A & \text{if } \tau(A) = *, \\ \forall x.rx \ \mathbf{mr}\ A, & \text{otherwise.} \end{cases}$$

$$r \ \mathbf{mr}\ \exists x\, A \quad := \quad \begin{cases} \epsilon \ \mathbf{mr}\ A[x/r] & \text{if } \tau(A) = *, \\ \pi_1(r) \ \mathbf{mr}\ A[x/\pi_0(r)] & \text{otherwise.} \end{cases}$$

Correctness of the Extracted Program

Theorem (Soundness Theorem)

*If d is a proof of a formula A, then we can derive $[\![d]\!]$ **mr** A from the assumptions $\{\bar{u}\colon x_u \textbf{ mr } C \mid u^C \in \mathrm{FA}(d)\}$, where $x_u := \epsilon$ if u^C is an assumption variable for a formula C without computational content.*

Proof.

By induction on the structure of d. (On board.) □

Theorem (Extraction Theorem)

From a proof d of $\forall x \exists y B(x, y)$, B computationally meaningless, from computationally meaningless assumptions $\{C\}$, one can extract a closed term $[\![d]\!]$ such that the formula $\forall x B(x, [\![d]\!])$ is provable from $\{C\}$.

Non-computational quantifiers

Idea: if in an *forall* introduction, apart from the usual variable condition, we know in addition that the variable is not used free in any term used in the proof so far, we mark this situation with a label nc to the formula. (Formal property, see slide 24. We briefly go through the definitions again add modify them accordingly.)

Definition (Formulas with nc-quantifiers)

In the definition of formulas we introduce two sorts of quantifiers, the usual quantifiers $\forall, \exists$ and quantifiers $\forall^{\mathsf{nc}}$, $\exists^{\mathsf{nc}}$ carrying **n**o computational **c**ontent.

Hence formulas are (extended to)

$$P(\vec{t}) \mid A \to B \mid A \land B \mid \forall x^\rho A \mid \forall^{\mathsf{nc}} x^\rho A \mid \exists^{\mathsf{nc}} x^\rho A \mid \exists^{\mathsf{nc}} x^\rho A$$

where in $P(\vec{t})$ we assume that P is of arity $\vec{\rho}$ and the terms $\vec{t} = t_1, \ldots, t_n$ are of types $\rho_1, \ldots, \rho_n$ respectively.

For formulas with a quantifier containing no computational content the obvious definition is

$$
\begin{aligned}
\tau(\forall^{\mathsf{nc}} x^{\rho}\, A) &:= \tau(A) \\
\tau(\exists^{\mathsf{nc}} x^{\rho}\, A) &:= \tau(A)
\end{aligned}
$$

Definition (Extracted Program incl nc quantifiers)

$$
\begin{aligned}
[\![(\lambda x d)^{\forall x A}]\!] &:= \lambda x [\![d]\!], \\
[\![d^{\forall x A} t]\!] &:= [\![d]\!] t.
\end{aligned}
$$

$$
\begin{aligned}
[\![(\lambda x d)^{\forall^{\mathrm{nc}} x A}]\!] &:= [\![d]\!], \\
[\![d^{\forall^{\mathrm{nc}} x A} t]\!] &:= [\![d]\!].
\end{aligned}
$$

Definition (Proofs incl nc quantifiers)

Proofs are presented as lambda terms via the Curry-Howard Correspondence.

$$
\begin{aligned}
& u^A \mid c^A \text{ (c an axiom)} \mid (\lambda u^A d^B)^{A \to B} \mid (d^{A \to B} e^A)^B \mid \\
& (\langle d^A, e^B \rangle)^{A \wedge B} \mid (\pi_0(d^{A \wedge B}))^A \mid \pi_1((d^{A \wedge B}))^B \mid \\
& (d^{\forall x A} t)^{A(t)} \mid (d^{\forall^{\mathrm{nc}} x A} t)^{A(t)} \mid \\
& (\lambda x d)^{\forall x A},\ x \notin \mathrm{FV}(C) \text{ for } u^C \in \mathrm{FA}(d) \mid \\
& (\lambda x d)^{\forall^{\mathrm{nc}} x A},\ x \notin \mathrm{CV}(d) \cup \mathrm{FV}(C) \text{ for } u^C \in \mathrm{FA}(d)
\end{aligned}
$$

where $\mathrm{CV}(d)$ is defined as follows:

If $\tau(A) \neq *$, then

(ass)	$\mathrm{FA}(u) := \{u\}$	$\mathrm{CV}(u) := \emptyset$
(ax)	$\mathrm{FA}(c) := \emptyset$	$\mathrm{CV}(c) := \emptyset$
$(\to^+)$	$\mathrm{FA}(\lambda u.d) := \mathrm{FA}(d) \setminus \{u\}$	$\mathrm{CV}(\lambda u.d) := \mathrm{CV}(d)$
$(\to^-)$	$\mathrm{FA}(de) := \mathrm{FA}(d) \cup \mathrm{FA}(e)$	$\mathrm{CV}(de) := \mathrm{CV}(d) \cup \mathrm{CV}(e)$
$(\wedge^+)$	$\mathrm{FA}(\langle d, e\rangle) := \mathrm{FA}(d) \cup \mathrm{FA}(e)$	$\mathrm{CV}(\langle d, e\rangle) := \mathrm{CV}(d) \cup \mathrm{CV}(e)$
$(\wedge^-)$	$\mathrm{FA}(\pi_i(d)) := \mathrm{FA}(d)$	$\mathrm{CV}(\pi_i(d)) := \mathrm{CV}(d)$
$(\forall^+)$	$\mathrm{FA}((\lambda x.d)^{\forall x A}) := \mathrm{FA}(d)$	$\mathrm{CV}((\lambda x.d)^{\forall x A}) := \mathrm{CV}(d) \setminus \{x\}$
$(\forall^{\mathsf{nc}+})$	$\mathrm{FA}((\lambda x.d)^{\forall^{\mathsf{nc}} x A}) := \mathrm{FA}(d)$	$\mathrm{CV}((\lambda x.d)^{\forall^{\mathsf{nc}} x A}) := \mathrm{CV}(d)$
$(\forall^-)$	$\mathrm{FA}(d^{\forall x A} t) := \mathrm{FA}(d)$	$\mathrm{CV}(d^{\forall x A} t) := \mathrm{CV}(d) \cup \mathrm{FV}(t)$
$(\forall^{\mathsf{nc}-})$	$\mathrm{FA}(d^{\forall^{\mathsf{nc}} x A} t) := \mathrm{FA}(d)$	$\mathrm{CV}(d^{\forall^{\mathsf{nc}} x A} t) := \mathrm{CV}(d)$

Otherwise, i.e., if $\tau(A) = *$, we set $\mathrm{CV}(d^A) := \emptyset$ and $\mathrm{FA}(d^A)$ is defined as above.

Definition (Modified Realizability) (cont)

$$r \ \mathbf{mr}\ \forall x A \quad := \quad \begin{cases} \forall x.\epsilon \ \mathbf{mr}\ A & \text{if } \tau(A) = *, \\ \forall x.rx \ \mathbf{mr}\ A, & \text{otherwise.} \end{cases}$$

$$r \ \mathbf{mr}\ \exists x\, A \quad := \quad \begin{cases} \epsilon \ \mathbf{mr}\ A[x/r] & \text{if } \tau(A) = *, \\ \pi_1(r) \ \mathbf{mr}\ A[x/\pi_0(r)] & \text{otherwise.} \end{cases}$$

In the case of quantifiers without computational content we set

$$\begin{aligned} r \ \mathbf{mr}\ \forall^{\mathrm{nc}} x\, A \quad &:= \quad \forall x.\ r \ \mathbf{mr}\ A \\ r \ \mathbf{mr}\ \exists^{\mathrm{nc}} x\, A \quad &:= \quad \exists x.\ r \ \mathbf{mr}\ A. \end{aligned}$$

DEMO (Interactive proof and extracted program
for quotient and remainder example)
in the interactive proof assistant Minlog
www.minlog-system.de

The interactive proof assistant: Minlog

- Formal system = Heyting Arithmetic in finite types HA^{ω}
  - = Functional term language with structural recursion
    Intuitionistic logic + Induction
  - + Classical logic via axioms
  - + Constants, free predicate variables,
  - + Inductive types, Inductively defined predicates
  - + New: extension to Coinduction
- Model: partial continuous functionals in finite types.
- Proofs are represented as lambda terms (Curry-Howard)
  - Can be checked,
  - normalized (Normalization-by-Evaluation)
  - manipulated for program development, etc
- Automatization available.

Inductive Definitions

Definition (Inductively defined predicates)

An inductively defined predicate $I : \rho_1, \ldots, \rho_l$ is introduced by n closure axioms, $K_1[I], \ldots, K_n[I]$,$1 \leq n$, (also called introduction axioms), where

$$K_i[I] := \forall x_i.A_i \to (\forall y_i.\, B_i \to I(s_i)) \to I(t_i).$$

Given a predicate P, let $K_i[P]$ be the formula which is obtained by replacing the predicate I in $K_i[I]$ by P. Then, the induction principle (also called elimination axiom) is

$$K_1[P] \to \ldots \to K_n[P] \to \forall^{\mathrm{nc}} \vec{z}.I(\vec{z}) \to P(\vec{z}).$$

Definition (The type of an inductively defined predicate)

In the general case of an inductive predicate I with computational content, given by the axioms $K_1[I], \dots, K_n[I]$ where

$$K_i[I] := \forall \vec{x_i}^{\vec{\rho_i}}, \forall^{\mathrm{nc}} \vec{x_i}'^{\vec{\rho_i}'} . \overrightarrow{A_i} \to \overrightarrow{\forall \vec{y_i}^{\vec{\sigma_i}}, \forall^{\mathrm{nc}} \vec{y_i}'^{\vec{\sigma_i}'} . \overrightarrow{B_i} \to I(\vec{s_i})} \to I(\vec{t_i}),$$

we set

$$\tau_0(I) := \mu,$$

where μ is either inductively defined by

$$\begin{array}{rcl} \mu & = & c_1(\vec{\rho_1},\, \tau(\overrightarrow{A_1}),\, \overrightarrow{\vec{\sigma_1} \to \tau(\overrightarrow{B_1}) \to \mu}) \\ & & + \cdots + \\ & & c_n(\vec{\rho_n},\, \tau(\overrightarrow{A_n}),\, \overrightarrow{\vec{\sigma_n} \to \tau(\overrightarrow{B_n}) \to \mu}) \end{array}$$

with new constructors $c_1, \dots, c_n$ or it is an existing inductive type with constructors of the same type. Here, we have written $\tau(\vec{A})$ for $\tau(\vec{A}_1), \dots, \tau(\vec{A}_{|\vec{A}|})$ and $\tau(\vec{B}) \to \mu$ for $\tau(\vec{B}_1) \to \cdots \to \tau(\vec{B}_{|\vec{B}|}) \to \mu$.

Given an inductively defined predicate I, we need realizers for the closure axioms $K_1[I], \dots, K_n[I]$ and the induction principle $\mathrm{Ind}_{I,P}$. Assume that we have assigned an algebra μ with constructors $c_1, \dots, c_n$ to this predicate, i.e., that we are dealing with a predicate with computational content. Then we set

$$[\![K_i[I]]\!] := c_i$$

and the induction principle corresponds to recursion on μ, more precisely,

$$[\![\mathrm{Ind}_{I,P}]\!] := \mathrm{R}_{\mu,\tau(P)}.$$

Proof.

Omitted. □

Another example: Reverse

Prove that every list can be reversed.

$$\text{Goal:} \quad \forall v \exists w \mathrm{Rev}(v, w)$$

where the predicate Rev is axiomatized by

$$\begin{aligned}
&\mathrm{Rev}(\mathrm{Nil}, \mathrm{Nil}) \\
&\mathrm{Rev}(v, w) \to \mathrm{Rev}(v :+: [a], a :: w)
\end{aligned}$$

Another example: Reverse

Prove that every list can be reversed.

$$\text{Goal:} \quad \forall v \exists w \mathrm{Rev}(v, w)$$

where the predicate Rev is axiomatized by

$$\mathrm{Rev}(\mathrm{Nil}, \mathrm{Nil})$$
$$\mathrm{Rev}(v, w) \to \mathrm{Rev}(v :+: [a], a :: w)$$

Proof: By induction on v. *Base:* Clear.
Step: Fix a,v and IH: $\exists w \mathrm{Rev}(v, w)$ and show $\exists w' \mathrm{Rev}(a :: v, w')$.
Solution: take $w' = w :+: [a]$.

Extracted MINLOG term

Reverse:=

```
((listrec |Nil|)
 (lambda (n1)
   (lambda (v2)
     (lambda (v3) ((|ListAppend| v3) ((|Cons| n1) |Nil|))))))
```

More readable as recursive equations:

$$
\begin{array}{llcl}
\text{Reverse} & \text{Nil} & = & \text{Nil} \\
\text{Reverse} & (\mathtt{Cons}\ n_1\ v_2) & = & (\text{Reverse}\ v_2) :+: (\mathtt{Cons}\ n_1\ \text{Nil})
\end{array}
$$

Extracted MINLOG term

Reverse:=

```
((listrec |Nil|)
 (lambda (n1)
   (lambda (v2)
     (lambda (v3) ((|ListAppend| v3) ((|Cons| n1) |Nil|))))))
```

More readable as recursive equations:

$$
\begin{array}{lcl}
\text{Reverse} & \text{Nil} & = & \text{Nil} \\
\text{Reverse} & (\mathtt{Cons}\ n_1\ v_2) & = & (\text{Reverse}\ v_2) :+: (\mathtt{Cons}\ n_1\ \text{Nil})
\end{array}
$$

Example: Reverse with a classical proof

Goal: $\forall v \exists^{\mathrm{cl}} w \mathrm{Rev}(v, w)$

Proof: Assume that there is a list v_0 which cannot be reversed and show a contradiction.

Then we can show that all initial segments of v_0 cannot be reversed either, i.e.

$$\forall u, v.\ \ v +\!\!+\, u = v_0 \quad \to \quad \forall w\ \neg \mathrm{Rev}(v, w).$$

By induction on u (using the assumption that v_0 cannot be reversed).
We get a contraction because Nil can be reversed.

Question: How does the program extracted from such a proof look?

A-Translation

Idea: Start with classical proof, G quantifier free.

$$
\begin{array}{rl}
 & \vdash_c \exists^{\mathrm{cl}} y G \\
\overset{\Longrightarrow}{\textit{Double Negation Translation}} & \vdash_m (\forall y.\, G^{\neg\neg} \to \bot) \to \bot. \\
\overset{\Longrightarrow}{\textit{Replace } \bot \textit{ by arbitrary } X} & \vdash_m (\forall y.\, (G^{\neg\neg})^X \to X) \to X. \\
\overset{\Longrightarrow}{\textit{Friedman } X := \exists y G} & \vdash_m (\forall y.\, (G^{\neg\neg})^{\exists y G} \to \exists y G) \to \exists y G. \\
\overset{\Longrightarrow}{\textit{Premise provable}} & \vdash \exists y G.
\end{array}
$$

A-translation = Double negation translation + Friedman Trick

Refined A-Translation

Refinement: allow assumptions D, let D, G be as general as possible, i.e. only do double negations where necessary.

Theorem (Berger, Buchholz, Schwichtenberg)

Let D be a definite formula, G be a goal formula:

$$\vdash_m D \to (\forall y.\, G \to \bot) \to \bot.$$

Then, we can extract

a program p and get a proof $\vdash D \to G[y/p]$

Definitions of definite and goal formulas on the next slide.

Definition (Relevant and irrelevant formulas)

A formula is relevant if it "ends" with $\bot$. More precisely, *(Ir)relevant formulas* are defined inductively by the clauses

- $\bot$ is relevant, all other atomic formulas are irrelevant,
- if C is (ir)relevant and B is arbitrary, then $B \to C$ is (ir)relevant,
- if, C_0 and C_1 are (ir)relevant, then $C_0 \wedge C_1$ is (ir)relevant.
- if C is (ir)relevant, then $\forall x C$ is (ir)relevant.

Definition (Definite and goal formulas)

$$
\begin{aligned}
D &:= P \mid G \to D \text{ (provided } D \text{ not rel. } \Rightarrow G \text{ irrel.)} \mid D \wedge D \mid \forall x D. \\
G &:= P \mid D \to G \text{ (prov. } D \text{ not rel.} \Rightarrow D \text{ dec.)} \mid G \wedge G \mid \forall x G \text{ (prov. } G \text{ irr.)}
\end{aligned}
$$

Extracted program from classical proof $\forall v \exists^{cl} w \mathrm{Rev}(v, w)$

Reverse:=

```
(lambda (v0)
 ((((listrec (lambda (v1) v1))
    (lambda (n1)
      (lambda (v2)
        (lambda (f3) (lambda (v4) (f3 (n1::v4)))))))
   v0)
  |Nil|))
```

More readable as recursive equations:
Reverse v_0 = reverse v_0 $\mathtt{Nil}$

$$\begin{array}{llcr} \text{reverse} & \mathtt{Nil}\ v_1 & = & v_1 \\ \text{reverse} & (n_1 :: v_2)\ v_4 & = & \text{reverse}\ v_2 (n_1 :: v_4) \end{array}$$

Linear instead of quadratic program!

Program extraction from classical proofs

Note this result is only possible if we make use of the $\forall^{\mathrm{nc}}$ quantifier. (Can be assigned automatically!)

Goal: $\forall v \exists^{\mathrm{cl}} w \mathrm{Rev}(v, w)$

Proof: Assume that there is a list v_0 which cannot be reversed and show a contraction.

Then we proved that all initial segments of v_0 cannot be reversed either, i.e.

$$\forall u,\ \forall^{\mathrm{nc}} v.\ v +\!\!+\, u = v_0 \ \to\ \forall w\ \neg \mathrm{Rev}(v, w).$$

The use of the (non-computational) nc-quantifier in the proof is essential!

Otherwise we would extract:

$\mathrm{Reverse}_2\, v_0 = \mathrm{reverse}_2\ v_0\ \mathtt{Nil}\ \mathtt{Nil}$

$$
\begin{array}{lrcl}
\mathrm{reverse}_2 & \mathtt{Nil}\, v_1\, v_2 & = & v_2 \\
\mathrm{reverse}_2 & (\mathtt{Cons}\, n_1\ v_2) v_4\ v_5 & = & \mathrm{reverse}_2\ v_2 \\
 & & & v_4 +\!\!+ [n_1] \\
 & & & (n_1 :: v_5)
\end{array}
$$

Application: Higman's Lemma

Definition (Wqo)

A quasiorder $(A, \leq_A)$ (i.e. $\leq_A$ refl and trans.) is a well quasiorder (wqo), if every infinite sequence of elements in A is good, i.e.,

$$\forall (a_i)_{i<\omega} \, \exists i, j. \, i < j \to a_i \leq_A a_j$$

Lemma (Higman's Lemma)

If $(A, \leq_A)$ is a wqo, then also the set of finite lists with element in A $(A^*, \leq_{A^*})$ is a wqo.

Proof: Use minimal-bad-sequence argument (Sketched on board).

Question: What is the program extracted from such a proof?

Demo in the interactive theorem prover Minlog; to understand extracted program we look at a simpler problem on the following slides.

Formal proof with Classical Dependent Choice

In the proof we used the following instance of dependent choice:

$$\mathrm{DC}' \quad B([]) \wedge \forall ns(B(ns) \to \exists n\, B(ns * n)) \to \exists g \forall k\, B(\bar{g}k)$$

DC′ follows from the more common scheme of DC:

$$\forall n \forall x^\rho \exists y^\rho A_n(x, y) \to \exists f^{\mathrm{nat} \to \rho}.\ f(0) = x_0^\rho \wedge \forall n A_n(f(n), f(n+1))$$

Why are choice principles problematic for A-translation?

- DC'^X is not an instance of DC' anymore [in contrast to induction axioms, for example]
- DC' is not a definite formula, nor can it be transformed into one using the usual double negation translation.

Solution: Define realizer directly for the A-translated version of the choice principle.

See Beradi, Bezem, Coquand, further developed by Berger, Oliva

A-Translation - extended.

What if we want to use assumptions which are neither definite, nor can be translated into a definite formula?

Theorem

Let A be an arbitrary formula, but assume that we have a system Δ and a term t such that

$$\Delta \vdash t \ \mathbf{mr} \ A^X.$$

Let D be a definite formula,
let G be a goal formula, and assume that

$$\vdash_m A \to D \to (\forall y.\, G \to \bot) \to \bot.$$

Then, there is a program p such that

$$\Delta \vdash D \to \ G[y/p]$$

Example: The infinite pigeon hole principle

Thm

Every boolean valued sequence $f^{\mathsf{nat}\to\mathsf{boole}}$ has a constant infinite subsequence, i.e. there are infinitely many indices where the sequence is either constant true or constant false.

Corollary (Finite Version)

For each boolean sequence $f^{\mathsf{nat}\to\mathsf{boole}}$ and each number n there is a constant subsequence of length n, i.e. there is a list of indices $ns = [n_0, \ldots n_{n-1}]$ and a boolean value a such that $\forall i < n.\, f(ns_i) = a$.

Example

$$\begin{array}{lcl} f & = & T\ F\ T\ T\ F\ T\ T\ F\ F\ \ldots \\ n & = & 4 \\ \\ ns = [n_0, \ldots, n_3] & = & [0, 2, 3, 6] \end{array}$$

A proof of the infinite pigeon principle using dependent choice.

Thm

Every boolean valued sequence $f^{\text{nat}\rightarrow\text{boole}}$ has a constant infinite subsequence, i.e. there are infinitely many indices where the sequence is constant.

Informal proof: One of the following two cases must hold:

1.
$$FTFFTF\dots TTTTTTTTTTTTT$$
The sequence f becomes constant T from a certain point on, then we have a subsequence which is always T.

2.
$$FTFFTFTTTFFTFTTTTT\ F\ TT\ F\ T\dots$$
Or $\forall x \exists y > x. f(y) = F$. Then, using classical dependent choice, we have a subsequence which is constant F.

Formal proof with Classical Dependent Choice

Formally we prove:

$$\forall f^{\text{nat}\to\text{boole}}\,\forall n\,\exists ns^{\text{list nat}}.\,|ns| = n \wedge \text{Inc}\,ns \;\wedge\; \text{Cst}\,f\,ns.$$

were $\text{Inc}\,ns$ (indices ns are increasing) and $\text{Cst}\,f\,ns$ (f is constant on ns) are defined appropriately.

We used the following instance of dependent choice:

$$\text{DC}' \quad B([]) \wedge \forall ns(B(ns) \to \exists n\, B(ns * n)) \to \exists g \forall k\, B(\bar{g}\,k)$$

with $B(ns) := \neg\neg(\text{Inc}\; ns \wedge \text{Cst}\,f\; ns)$.
$*$ denotes the cons operation on lists.

Realizer for DC'

Theorem

Using modified bar recursion, define $\Psi_{G_0,G,Y}(t) :=$

$$\tilde{Y}(t\#\lambda n.\langle 0^\rho, H(G(\pi_0 \circ t, ([G_0]\#(\pi_1 \circ t))_{|t|}, \lambda x^\rho, z^\sigma.\Psi_{G_0,G,Y}(t * \langle x, z\rangle))))\rangle)$$

where

- $\tilde{Y}(\beta) := Y(\pi_0 \circ \beta, [G_0]\#(\pi_1 \circ \beta))$
- H *realizes* $X \to B^X$.
- $\#$ *denotes concatenation of lists (2.arg. poss. inf.)*
- π_0/π_1 *are used for projections.*

Then

$$\lambda G_0, G, Y.\Psi_{G_0,G,Y}[] \ \mathbf{mr} \ DC'^X$$

where (DC') is used with a relevant formula B.

Realizer for DC': Axioms needed

Let $\#$ denote concatenation, $*$ concatenation of one elt.

1. *Modified bar recursion at type* ρ with the defining equation:

$$\Psi(Y, G, s) \stackrel{\tau}{=} Y(s\#\lambda k.G(k, s, \lambda x.\Psi(Y, G, s * x))).$$

2. *Principle of continuity*:
   $\forall F^{(\mathrm{nat}\to\rho)\to\tau}, \alpha^{\mathrm{nat}\to\rho}\exists n\forall\beta(\bar{\alpha}n{=}\bar{\beta}n \to F(\alpha){=}F(\beta)).$
3. *Principle of relativised quantifier free bar induction*

$$\frac{\begin{array}{c}\forall\alpha \in S\exists nP(\bar{\alpha}n)\\ \forall s \in S.\forall x(S(s * x) \to P(s * x)) \to P(s)\\ S([])\end{array}}{P([])}$$

```
(lambda (f0)
  (lambda (n1)
    ((((|Psi| (lambda (ns2) ns2))
        (lambda (ns2)
          (lambda (z3)
            (lambda (|(nat=>(tsil nat=>tsil nat)=>tsil nat)_4|)
              ((((natrec n1) (lambda (ns5) ns5))
                 (lambda (n5)
                   (lambda (z6)
                     (lambda (ns7)
                       (if (f0 ((|natPlus| (|Next| ns2)) n5))
                           (z6 ns7)
                           (z3 ((|(nat=>(tsil nat=>tsil nat)=>tsil nat)_4|
                                  ((|natPlus| (|Next| ns2)) n5))
                                 (lambda (ns8) ns8))))))))
                ((fbar (|natPlus| (|Next| ns2))) n1))))))
       (lambda (e2)
         (lambda (|(nat=>tsil nat=>tsil nat)_3|)
           ((|(nat=>tsil nat=>tsil nat)_3| n1) ((fbar e2) n1)))))
      |Lin|)))
```

Extracted term in Minlog: $n = 3$ and general case

$n = 3$:

```
f = TTTTTTTTT...

ns = [0,1,2]

f = FFFFFFFFFFFF...

ns = [2,5,8]
```

General case: the programs finds

- either n connected indices belonging to T's,
- or n non-connected indices belonging to F's.

Discuss efficiency Scheme vs Haskell.

Importance of evaluation strategy: Scheme vs Haskell

If $n = 5$, which indices would be found by the following function?

F	$TTT\ldots$	F	$T\ T\ \ldots$	F	$\ldots$
0	1 2 3 4 5 6 7 8 9	10	11 12 13 14 15 16 17 18 19	20	$\ldots$

$ns = [1, 2, 3, 4, 5]$

F	$TTTT$	F	$T\ldots$	F	$T\ T\ \ldots$	F	$\ldots$
0	1 2 3 4	5	6 7 8 9	10	11 12 13 14 15 16 17 18 19	20	$\ldots$

$ns = [11, 12, 13, 14, 15]$

Number of calls to f? :
In Haskell: 12 - called at: 4 3 2 1 0 5 10 15 14 13 12 11
In Scheme: 1980.

Extraction of a SAT solving algorithm

Basic definitions:

- A *literal* l is either a positive variable $+v$ or a negative variable $-v$. The *opposite* value of a literal is defined as: $\bar{+}v = -v$, $\bar{-}v = +v$.
- A *clause* C is defined as a set of literals $\{l_1, \ldots, l_k\}$ (representing their disjunction).
- A *formula* is a set of clauses (representing their conjunction).

An example of a formula:

$$\{\{l_{11}\}, \{l_{21}\}, \{\neg l_{11}, \neg l_{21}\}\}$$

to be read as

$$l_{11} \wedge l_{21} \wedge \{\neg l_{11} \vee \neg l_{21}\}$$

SAT problem: is there a valuation for these variables satisfying the formula?

The need for verified SAT algorithms.

* SAT algorithms are both part of safety critical software and also used for the verification and certification thereof.
* They are nowadays highly optimized for speed, which makes the introduction of errors more likely and their verification more difficult.
* Beside correctness also totality is an issue. Eg 2012 SAT competition www.smt.org many systems were not total, returning "unknown" for certain inputs.

DPLL and Related Work

Most modern SAT solvers are based on the DPLL algorithm (Davies, Putnam, Logemann, Loveland 1960/1962)
The DPLL algorithm has been verified in both Coq and Isabelle.

Both of these approaches formally state the algorithm before verifying it. However, in contrast to this, the algorithm can also be extracted.

DPLL Proof System

The DPLL proof system is defined by an inductive definition and proves unsatisfiability:

$$\frac{\Gamma, l \vdash \Delta}{\Gamma \vdash \Delta, \{l\}}\,(\mathit{Unit}) \qquad \frac{\Gamma, l \vdash \Delta, C}{\Gamma, l \vdash \Delta, (\bar{l}, C)}\,(\mathit{Red})$$

$$\frac{\Gamma, l \vdash \Delta}{\Gamma, l \vdash \Delta, (l, C)}\,(\mathit{Elim})$$

$$\frac{}{\Gamma \vdash \Delta, \emptyset}\,(\mathit{Conflict}) \qquad \frac{\Gamma, l \vdash \Delta \qquad \Gamma, \bar{l} \vdash \Delta}{\Gamma \vdash \Delta}\,(\mathit{Split})$$

Here Γ is a valuation (set of literals, with values already assigned) and Δ is a formula (clause set) .

Valuations and Models

- A *valuation* Γ, i.e., set of literals $\{l_1, \ldots, l_k\}$ is *consistent* iff $l \in \Gamma \to \bar{l} \notin \Gamma$. Let Cons be the set of all consistent Valuations.
- A *model* is a total function M which maps literals to booleans and satisfies the following property $\forall l, M.\ Ml \leftrightarrow \neg(M\ \bar{l})$

Two abbreviations:

- For a given valuation Γ, $\forall l \in \Gamma\ M\ l$ is abbreviated as $M \models \Gamma$.
- For a given formula Δ, $\forall C \in \Delta\ \exists l \in C\ M\ l$ is abbreviated as $M \models \Delta$.

We call a valuation Γ and a formula Δ *compatible* if there exists a model satisfying both, i.e.

$$\exists M.\, M \models \Gamma \wedge M \models \Delta,\ \ i.e.$$
$$\exists M.\, M \models \Gamma \wedge \forall C \in \Delta\ \exists l \in C\ M\ l.$$

Formalising and Proving Completeness

The expected statement of completeness is: $\forall \Gamma \in \mathrm{Cons}, \forall \Delta.$

$$\mathrm{incompatible}(\Gamma; \Delta) \to \Gamma \vdash \Delta$$

We proved the following classically equivalent but constructively stronger statement: $\forall \Gamma \in \mathrm{Cons}, \forall \Delta.$

$$\mathit{compatible}(\Gamma; \Delta) \vee \Gamma \vdash \Delta$$

Formalising and Proving Completeness

The expected statement of completeness is: $\forall \Gamma \in \text{Cons}, \forall \Delta.$

$$\text{incompatible}(\Gamma; \Delta) \to \Gamma \vdash \Delta$$

We proved the following classically equivalent but constructively stronger statement: $\forall \Gamma \in \text{Cons}, \forall \Delta.$

$$\textit{compatible}(\Gamma; \Delta) \vee \Gamma \vdash \Delta$$

Program extraction yields a program that either yields a model if Γ and Δ are compatible ($\exists M.\, M \models \Gamma \wedge M \models \Delta$) or a derivation if incompatible.

Proof of Completeness Theorem

Theorem: $\forall \Gamma \in \text{Cons}, \forall \Delta, \Theta.\ \emptyset \notin \Theta \wedge \mathit{Var}(\Gamma) \cap \mathit{Var}(\Theta) = \emptyset \rightarrow$

$$(\Gamma \vdash \Delta \cup \Theta) \ \vee \ \exists M.\, M \models \Gamma \wedge M \models \Delta \cup \Theta,$$

We aim to perform the proof in such a way that an efficient program is extracted:

1. Since performing a split is the only computational expensive operation, we only apply it when it is absolutely necessary.
2. We perform an optimization on the proof level by partitioning the clauses into 'clean' and 'unclean' clauses, where a clause is called clean if we cannot apply Elim, Reduce or Unit to that clause.

Extracted Solver

We run the extracted solver using *pigeon hole formulae*

$$PHP(n, m) := \{l_{i,1} \vee \ldots \vee l_{i,m} | 1 \leq i \leq n\}$$
$$\cup \{\neg l_{i,k} \vee \neg l_{j,k} | 1 \leq i < j \leq n, 1 \leq k \leq m\}$$

Intuitively, $PHP(n, m)$ states "it is not possible to put n pigeons into m holes and only have one pigeon in each hole"

Extracted Program (cont.)

On satisfiable formulae:

$PHP(2,2)$	$PHP(3,3)$	$PHP(4,4)$	$PHP(5,5)$	$PHP(6,6)$
$<$ 1 Sec	$<$ 1 Sec	5.45	26.09	1:34.11

On unsatisfiable formulae:

$PHP(2,1)$	$PHP(3,2)$	$PHP(4,3)$	$PHP(5,4)$	$PHP(6,5)$
$<$ 1 Sec	1.17	33.62	13:54	5:35:41

Extraction to Haskell:

Formula	Minlog $\forall$	Minlog $\forall^{nc}$	Haskell		Haskell (`-fllvm`)	
	Witness	Witness	Witness	Yes/No	Witness	Yes/No
PHP(4,3)	33.62s	11.61s	0.019s	0.006s	0.015s	0.004s
PHP(4,4)	5.45s	5.25s	0.019s	0.010s	0.014s	0.007s
PHP(5,4)	13m54s	2m41s	0.055s	0.020s	0.036s	0.012s
PHP(5,5)	26.09s	25.03s	0.024s	0.015s	0.020s	0.010s
PHP(6,5)	5h35m41s	37m25s	0.367s	0.066s	0.279s	0.039s
PHP(6,6)	1m34.11s	1m24.88s	0.035s	0.025	0.025s	0.015s
PHP(8,8)	-	-	0.054s	0.029s	0.040s	0.025s
PHP(9,8)	-	-	-	1m21.915s	-	32.062s
PHP(9,9)	-	-	0.064s	0.042s	0.052s	0.030s
PHP(10,9)	-	-	-	102m 16s	-	15m 5s

Performance compared to Versat

Versat was formalized and verified in the dependently typed programming language Guru and translated into C-code.

Formula	$\forall^{nc}$ compiled (Yes/No)	Versat
PHP(7,6)	0.226s	0.089s
PHP(8,7)	2.42s	0.794s
PHP(9,8)	32.062s	17.217s
PHP(10,9)	15m 5s	15m 46s

Comparison to Industrial Tool

Case study on the Verification of Railway Interlockings.
Contains 14726 clauses and 8166 variables.
Verified 109 safety conditions.
The hardest, called SC1, is unsatisfiable (due to the underspecification of the system which does not include all physical invariants).

Formula	$\forall^{nc}$ compiled		SCADE
	Yes/No	Witness	SAT/Counter Example
SC1	8s	12s	$<$1s

Program Extraction - Research Agenda

Establish Program Extraction from Formal Proofs as a Competitive Method for Program Development and Verification

The concrete steps are:

- Extend theory to cover a large range of proofs ($\rightarrow$ Inductive and coinductive Defs, Classical reasoning, Imperative programs, Higher Order Logic)
- Use it to reveal unknown computational content in proofs (Mathematics), but also apply to industrial problems.
- Highlight advantages over other methods of program development and verification: get algorithms for free.
- Improve feasibility; provide suitable tool support.

- G. Kreisel. On weak completeness of intuitionistic predicate logic. *The Journal of Symbolic Logic*, 27:139–158, 1962.
- A. S. Troelstra. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*, volume 344 of *LNM*. Springer, 1973.
- H. Schwichtenberg and S. Wainer, Proofs and Computations, Perspectives in Logic, Association for Symbolic Logic and Cambridge University Press, 2012.
- U. Berger, W. Buchholz, and H. Schwichtenberg, Refined Program Extraction from Classical Proofs, APAL 114:3–25, 2002.
- U. Berger, K. Miyamoto, M. Seisenberger, and H. Schwichtenberg, Minlog - A tool for program extraction supporting algebras and coalgebras, LNCS 6859, pp. 393-399, 2011.
- L. Crosilla, H. Schwichtenberg, M. Seisenberger, A Tutorial for Minlog,Version 5.0, available at www.minlog-system.de
- M. Seisenberger, Programs from Proofs using Classical Dependent Choice. APAL 153:97–110, 2008.
- U. Berger, A. Lawrence, F. Nordvall Forsberg, and M. Seisenberger, Extracting Verified Decision Procedures: DPLL and Resolution, LMCS 11(1), 2015.