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 $A \land B$ 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 $A \lor B$ 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 $A \rightarrow B$ is given by a construction or method that transforms a proof of $A$ into a proof of $B$.

$\neg A$ is treated like $A \rightarrow \bot$. 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)

\[ \forall a, b. \ 0 < b \rightarrow \exists q, r. \ a = q \times b + r \land r < b. \]

Proof: Ind(a). Base \( a = 0 \): take \( q = 0, \ r = 0 \).
Step: Given \( q, r \) such that \( a = q \times b + r \) and \( r < b \), find new \( q', r' \) for \( a + 1 \)
A first example

Example (Quotient and Remainder)

\[ \forall a, b. \ 0 < b \rightarrow \exists q, r. \ a = q \times b + r \land r < b. \]

Proof: Ind(a). Base \( a = 0 \): take \( q = 0, \ r = 0 \).
Step: Given \( q, r \) such that \( a = q \times b + r \) 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 \(\rightarrow\), that is, if \(\rho\) and \(\sigma\) are types, then so are \(\rho \times \sigma\) and \(\rho \rightarrow \sigma\); in short: types are

\[ \mu \mid \rho \times \sigma \mid \rho \rightarrow \sigma.\]

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.]
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 $\mu$ is introduced by the following equation:

$$
\mu = c_1(\vec{\rho}_1, \sigma_{11} \to \mu, \ldots, \sigma_{1m_1} \to \mu) \\
+ \cdots + \\
+ c_n(\vec{\rho}_n, \sigma_{n1} \to \mu, \ldots, \sigma_{nm_n} \to \mu)
$$

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, $\mu$ is the type whose elements are generated from the constructors

$$c_i : \vec{\rho}_i \to \vec{\sigma}_i \to \mu \to \mu.$$
Extended Heyting Arithmetic (cont.)

Definition (Terms)
are built from typed variables and constants via $\lambda$-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 $\mu$ and type $\tau$ we have a recursion operator $R_{\mu,\tau}$. If $\vec{\rho}_i \rightarrow \vec{\sigma}_i \rightarrow \mu \rightarrow \mu$ is the type of the $i$-th constructor $c_i$ of $\mu$, then the $i$-th step type $\delta_i$ is $\vec{\rho}_i \rightarrow \vec{\sigma}_i \rightarrow \mu \rightarrow \vec{\sigma}_i \rightarrow \tau \rightarrow \tau$ and the recursion operator has the type

$$R_{\mu,\tau} : \delta_1 \rightarrow \cdots \rightarrow \delta_n \rightarrow \mu \rightarrow \tau.$$

Analogously, we also have a case distinction operator

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

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

The conversion rules are

\[(\lambda x t)s \mapsto t[x/s]\]
\[(\lambda x t)x \mapsto t, \quad x \notin \text{FV}(t)\]
\[\pi_i(\langle t_0, t_1 \rangle) \mapsto t_i, \quad i = 0, 1\]
\[\langle \pi_0(t), \pi_1(t) \rangle \mapsto t\]

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

\[R_{\mu, \tau}\vec{s}(c_i \vec{t}) \mapsto s_i \vec{t}(R_{\mu, \tau}\vec{s} \circ t_1^R) \ldots (R_{\mu, \tau}\vec{s} \circ t_n^R)\]

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

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 $\bot$ 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

$$P(\vec{t}) \mid A \rightarrow B \mid A \land B \mid \forall x^\rho A \mid \exists x^\rho A,$$

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 $\ast$ (for “not computationally meaningful”).

\[
\tau(P(t)) := \begin{cases} 
\tau_0(P) & \text{if } P \in \mathcal{P} \text{ with assigned } \tau_0(P) \\
\ast & \text{otherwise.}
\end{cases}
\]

\[
\tau(A \rightarrow B) := \begin{cases} 
\tau(B) & \text{if } \tau(A) = \ast, \\
\ast & \text{if } \tau(B) = \ast, \\
\tau(A) \rightarrow \tau(B) & \text{otherwise.}
\end{cases}
\]

\[
\tau(A_0 \land A_1) := \begin{cases} 
\tau(A_i) & \text{if } \tau(A_{1-i}) = \ast, \\
\tau(A_0) \times \tau(A_1) & \text{otherwise.}
\end{cases}
\]

\[
\tau(\forall x^\rho A) := \begin{cases} 
\ast & \text{if } \tau(A) = \ast, \\
\rho \rightarrow \tau(A) & \text{otherwise.}
\end{cases}
\]

\[
\tau(\exists x^\rho A) := \begin{cases} 
\rho & \text{if } \tau(A) = \ast, \\
\rho \times \tau(A) & \text{otherwise.}
\end{cases}
\]
Part 1: Extraction of Programs from Constructive Proofs

Natural Deduction

Definition (Proofs)

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

\[ u^A \mid c^A \text{ (c an axiom)} \]
\[ \mid (\lambda u^A d^B)^{A\rightarrow B} \mid (d^{A\rightarrow B} e^A)^B \mid \]
\[ (\langle d^A, e^B \rangle)^{A\land B} \mid (\pi_0(d^{A\land B}))^A \mid (\pi_1(d^{A\land B}))^B \mid \]
\[ (d^{\forall x A} t)^{A(t)} \mid (\lambda x d)^{\forall x A}, \ x \notin \text{FV}(C) \text{ for } u^C \in \text{FA}(d) \]

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

1) For existential quantifier:

\[ (\exists^+_x A) \quad \forall x. A \rightarrow \exists x A \]
\[ (\exists^-_x A, B) \quad \exists x A \rightarrow (\forall x. A \rightarrow B) \rightarrow B, \quad (x \notin FV(B)) \]

2) Logical axioms Truth: \( T \) and efq-axioms \( \bot \rightarrow A \) for any formula \( A \).

3) Axioms for equality, such as reflexivity, transitivity, symmetry and compatibility, e.g.,

\[ (\text{Compat}) \quad \forall x, y. x = y \rightarrow P(x) \rightarrow P(y). \]

4) For each algebra, \( \mu \), 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:

$$(\exists^+_x, A) \quad \forall x. A \rightarrow \exists x A$$

$$(\exists^-_{x,A,B}) \quad \exists x A \rightarrow (\forall x. A \rightarrow B) \rightarrow B, \quad (x \not\in \text{FV}(B))$$

$$(\exists^{nc+}_{x,A}) \quad \forall^{nc} x. A \rightarrow \exists^{nc} x A$$

$$(\exists^{nc-}_{x,A,B}) \quad \exists^{nc} x A \rightarrow (\forall^{nc} x. A \rightarrow B) \rightarrow B, \quad (x \not\in \text{FV}(B))$$
Given a derivation $d$ of a computationally meaningful formula $A$, we inductively define the extracted term ("program") $[d]$ of type $\tau(A)$.

\[
[u^A] := x_\tau^A(A) \text{ (preassigned)}
\]

\[
[\lambda u^A d] := \begin{cases} 
[d] & \text{if } \tau(A) = * , \\
\lambda x^{\tau(A)}_u [d] & \text{otherwise}.
\end{cases}
\]

\[
d^{A \rightarrow B} e := \begin{cases} 
[d] & \text{if } \tau(A) = * , \\
[d][e] & \text{otherwise}.
\end{cases}
\]

\[
\langle d_0^A, d_1^A \rangle := \begin{cases} 
[d_i] & \text{if } \tau(A_{1-i}) = * \\
\langle [d_0], [d_1] \rangle & \text{otherwise}.
\end{cases}
\]

\[
[\pi_i(d_0^A \land A_1^A)] := \begin{cases} 
[d] & \text{if } \tau(A_{1-i}) = * \\
\pi_i[d] & \text{otherwise}.
\end{cases}
\]

\[
[(\lambda xd)^\forall xA] := \lambda x[d], \\
[d^\forall xA t] := [d] t.
\]
Extracted terms (realizers) for axioms

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

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

$$\left[ \exists^-_{x^\rho, A, B} \right] := \begin{cases} \lambda x^\rho \lambda f^{\tau(B)} f x & \text{if } \tau(A) = *, \\ \lambda z^\rho \times^{\tau(A)} \lambda f^{\tau(A) \rightarrow \tau(B)} f_0(z) \pi_1(z) & \text{otherwise.} \end{cases}$$

2) For a formula $A$ with $\tau(A) \neq *$, the efq axioms $F \rightarrow A$ and $\bot \rightarrow A$ are realized by a canonical inhabitant of the type $\tau(A)$.

3) The compatibility axiom, $\forall^{nc} x, y. x = y \rightarrow A(x) \rightarrow A(y)$, is realized by $\lambda x^{\tau(A)} x$.

4) Induction and case distinction on an inductive datatype $\mu$, $\text{Ind}_{\mu, A}$ and $\text{Cases}_{\mu, A}$, correspond to recursion on this datatype, $R_{\mu, \tau(A)}$, case distinction, $C_{\mu, \tau(A)}$, respectively.
Modified Realizability (Kreisel)

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

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

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

[For our first example, we do no have any atomic formulas/predicates with computational content.]
Part 1: Extraction of Programs from Constructive Proofs

Modified Realizability (cont)

\[ r \, \text{mr} \ (A \to B) := \begin{cases} \varepsilon \, \text{mr} \ A \to r \, \text{mr} \ B & \text{if } \tau(A) = *, \\ \forall x. \text{mr} \ A \to \varepsilon \, \text{mr} \ B & \text{if } \tau(A) \neq * = \tau(B), \\ \forall x. \text{mr} \ A \to r x \, \text{mr} \ B & \text{otherwise.} \end{cases} \]

\[ r \, \text{mr} \ (A_0 \land A_1) := \begin{cases} r \, \text{mr} \ A_{1-i} \land \varepsilon \, \text{mr} \ A_i & \text{if } \tau(A_i) = *, \\ \pi_0(r) \, \text{mr} \ A_0 \land \pi_1(r) \, \text{mr} \ A_1 & \text{otherwise.} \end{cases} \]

\[ r \, \text{mr} \ \forall x A := \begin{cases} \forall x. \varepsilon \, \text{mr} \ A & \text{if } \tau(A) = *, \\ \forall x. r x \, \text{mr} \ A, & \text{otherwise.} \end{cases} \]

\[ r \, \text{mr} \ \exists x A := \begin{cases} \varepsilon \, \text{mr} \ A[x/r] & \text{if } \tau(A) = *, \\ \pi_1(r) \, \text{mr} \ A[x/\pi_0(r)] & \text{otherwise.} \end{cases} \]
Correctness of the Extracted Program

**Theorem (Soundness Theorem)**

If $d$ is a proof of a formula $A$, then we can derive $\lbrack d \rbrack \text{ mr } A$ from the assumptions $\{ \bar{u} : x_u \text{ mr } C \mid u^C \in FA(d) \}$, 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 $\lbrack d \rbrack$ such that the formula $\forall x B(x, \lbrack d \rbrack)$ 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 $\forall^{nc}, \exists^{nc}$ carrying no computational content.

Hence formulas are (extended to)

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

where in $P(\vec{t})$ we assume that $P$ is of arity $\rho$ 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

\[
\tau(\forall^{nc} x^\rho A) := \tau(A)
\]

\[
\tau(\exists^{nc} x^\rho A) := \tau(A)
\]
Definition (Extracted Program incl nc quantifiers)

\[
\begin{align*}
\llbracket (\lambda x.d)^{\forall x.A} \rrbracket & := \lambda \llbracket d \rrbracket, \\
\llbracket d^{\forall x.A} t \rrbracket & := \llbracket d \rrbracket t.
\end{align*}
\]

\[
\begin{align*}
\llbracket (\lambda x.d)^{\forall^{nc} x.A} \rrbracket & := \llbracket d \rrbracket, \\
\llbracket d^{\forall^{nc} x.A} t \rrbracket & := \llbracket d \rrbracket.
\end{align*}
\]
Definition (Proofs incl nc quantifiers)

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

\[ u^A | c^A \quad (c \text{ an axiom}) \quad | \quad (\lambda u^A d^B)^{A\rightarrow B} \quad | \quad (d^{A\rightarrow B} e^A)^B \quad | \quad
\]
\[ (\langle d^A, e^B \rangle)^{A\wedge B} \quad | \quad (\pi_0(d^{A\wedge B}))^A \quad | \quad \pi_1((d^{A\wedge B}))^B \quad | \quad
\]
\[ (d^{\forall x A} t)^A(t) \quad | \quad (d^{\forall^{nc} x A} t)^A(t) \quad |
\]
\[ (\lambda xd)^{\forall x A}, \ x \notin \text{FV}(C) \text{ for } u^C \in \text{FA}(d) \quad | \quad
\]
\[ (\lambda xd)^{\forall^{nc} x A}, \ x \notin \text{CV}(d) \cup \text{FV}(C) \text{ for } u^C \in \text{FA}(d)
\]

where \( \text{CV}(d) \) is defined as follows:
If $\tau(A) \neq *$, then

- **(ass)** $\text{FA}(u) := \{u\}$  
- **(ax)** $\text{FA}(c) := \emptyset$  
- **($\rightarrow^+$)** $\text{FA}(\lambda u. d) := \text{FA}(d) \setminus \{u\}$  
- **($\rightarrow^-$)** $\text{FA}(de) := \text{FA}(d) \cup \text{FA}(e)$  
- **($\land^+$)** $\text{FA}(⟨d, e⟩) := \text{FA}(d) \cup \text{FA}(e)$  
- **($\land^-$)** $\text{FA}(\pi_i(d)) := \text{FA}(d)$  
- **($\forall^+$)** $\text{FA}(⟨\lambda x. d⟩^{\forall x A}) := \text{FA}(d)$  
- **($\forall^{nc+}$)** $\text{FA}(⟨\lambda x. d⟩^{\forall^{nc} x A}) := \text{FA}(d)$  
- **($\forall^-$)** $\text{FA}(d^{\forall x A} t) := \text{FA}(d)$  
- **($\forall^{nc-}$)** $\text{FA}(d^{\forall^{nc} x A} t) := \text{FA}(d)$  

$\text{CV}(u) := \emptyset$  
$\text{CV}(c) := \emptyset$  
$\text{CV}(\lambda u. d) := \text{CV}(d)$  
$\text{CV}(de) := \text{CV}(d) \cup \text{CV}(e)$  
$\text{CV}(⟨d, e⟩) := \text{CV}(d) \cup \text{CV}(e)$  
$\text{CV}(\pi_i(d)) := \text{CV}(d)$  
$\text{CV}(⟨\lambda x. d⟩^{\forall x A}) := \text{CV}(d) \setminus \{x\}$  
$\text{CV}(⟨\lambda x. d⟩^{\forall^{nc} x A}) := \text{CV}(d)$  
$\text{CV}(d^{\forall x A} t) := \text{CV}(d) \cup \text{FV}(t)$  
$\text{CV}(d^{\forall^{nc} x A} t) := \text{CV}(d)$

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

\[ r \, mr \, \forall x A := \begin{cases} \forall x. e \, mr \, A & \text{if } \tau(A) = *, \\ \forall x. r x \, mr \, A, & \text{otherwise.} \end{cases} \]

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

In the case of quantifiers without computational content we set

\[ r \, mr \, \forall^{nc} x A := \forall x. r \, mr \, A \]
\[ r \, mr \, \exists^{nc} x A := \exists x. r \, mr \, A. \]
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 $HA^\omega$
  - 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.
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

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

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

$$K_1[P] \rightarrow \ldots \rightarrow K_n[P] \rightarrow \forall^{nc} \vec{z}. I(\vec{z}) \rightarrow P(\vec{z}).$$
Definition (The type of an inductively defined predicate)

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

\[
K_i[I] := \forall \vec{x}_i^{\vec{\rho}_i}, \forall \text{nc} \vec{x}'_i^{\vec{\rho}'_i}, \vec{A}_i \rightarrow \forall \vec{y}_i^{\vec{\sigma}_i}, \forall \text{nc} \vec{y}'_i^{\vec{\sigma}'_i}, \vec{B}_i \rightarrow I(\vec{s}_i) \rightarrow I(\vec{t}_i),
\]

we set

\[
\tau_0(I) := \mu,
\]

where \( \mu \) is either inductively defined by

\[
\mu = \underbrace{c_1(\vec{\rho}_1, \tau(\vec{A}_1), \vec{\sigma}_1 \rightarrow \tau(\vec{B}_1) \rightarrow \mu)}_{+ \cdots +} + \underbrace{c_n(\vec{\rho}_n, \tau(\vec{A}_n), \vec{\sigma}_n \rightarrow \tau(\vec{B}_n) \rightarrow \mu)}_{+ \cdots +}
\]

with new constructors \( c_1, \ldots, c_n \) or it is an existing inductive type with constructors of the same type. Here, we have written \( \tau(\vec{A}) \) for \( \tau(\vec{A}_1), \ldots, \tau(\vec{A}|\vec{A}|) \) and \( \tau(\vec{B}) \rightarrow \mu \) for \( \tau(\vec{B}_1) \rightarrow \cdots \rightarrow \tau(\vec{B}|\vec{B}|) \rightarrow \mu \).
Given an inductively defined predicate $I$, we need realizers for the closure axioms $K_1[I], \ldots, K_n[I]$ and the induction principle $\text{Ind}_{I,P}$. Assume that we have assigned an algebra $\mu$ with constructors $c_1, \ldots, c_n$ to this predicate, i.e., that we are dealing with a predicate with computational content. Then we set

$$\llbracket K_i[I] \rrbracket := c_i$$

and the induction principle corresponds to recursion on $\mu$, more precisely,

$$\llbracket \text{Ind}_{I,P} \rrbracket := R_{\mu,\tau}(P).$$

**Proof.**
Omitted.
Another example: Reverse

Prove that every list can be reversed.

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

where the predicate Rev is axiomatized by

\[
\begin{align*}
\text{Rev}(\text{Nil, Nil}) \\
\text{Rev}(v, w) \rightarrow \text{Rev}(v : + : [a], a :: w)
\end{align*}
\]
Another example: Reverse

Prove that every list can be reversed.

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

where the predicate Rev is axiomatized by

\[
\begin{align*}
\text{Rev}(\text{Nil}, \text{Nil}) \\
\text{Rev}(v, w) \rightarrow \text{Rev}(v \:: + : [a], a :: w)
\end{align*}
\]

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

Reverse:=

\[
\text{((listrec |Nil|)} \\
\text{(lambda (n1)} \\
\text{(lambda (v2)} \\
\text{(lambda (v3) ((|ListAppend| v3) ((|Cons| n1) |Nil|))))))}
\]

More readable as recursive equations:

\[
\begin{align*}
\text{Reverse Nil} & = \text{Nil} \\
\text{Reverse (Cons } n_1 \text{ v2) } & = (\text{Reverse v2}): + : (\text{Cons } n_1 \text{ Nil})
\end{align*}
\]
Extracted **MINLOG** term

\[
\text{Reverse} := \\
((\text{listrec } |\text{Nil}|) \\
(\lambda (n1) \\
 (\lambda (v2) \\
 (\lambda (v3) ((|\text{ListAppend}| v3) ((|\text{Cons}| n1) |\text{Nil}|))))))
\]

More readable as recursive equations:

\[
\text{Reverse Nil} = \text{Nil} \\
\text{Reverse (Cons } n_1 \text{ v}_2 \text{)} = (\text{Reverse } v_2) : + : (\text{Cons } n_1 \text{ Nil})
\]
Example: Reverse with a classical proof

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

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.

\[ \forall u, v. \ v \mathbin{\mathbf{++}} u = v_0 \rightarrow \forall w \neg \text{Rev}(v, w). \]

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.

\[ \vdash_c \exists^c y G \]

Double Negation Translation

\[ \vdash_m (\forall y. G \neg
\neg \rightarrow \bot) \rightarrow \bot. \]

Replace $\bot$ by arbitrary $X$

\[ \vdash_m (\forall y. (G \neg
\neg)^X \rightarrow X) \rightarrow X. \]

Friedman $X := \exists y G$

\[ \vdash_m (\forall y. (G \neg
\neg)^\exists y G \rightarrow \exists y G) \rightarrow \exists y G. \]

Premise provable

\[ \vdash \exists y G. \]

A-translation $= \text{Double negation translation } + \text{Friedman Trick}$
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:

$$\vdash_m D \rightarrow (\forall y. G \rightarrow \bot) \rightarrow \bot.$$  

Then, we can extract

$$a \text{ program } p \quad \text{and get a proof } \vdash D \rightarrow G[y/p]$$

Definitions of definite and goal formulas on the next slide.
Definition ( Relevant and irrelevant formulas )
A formula is relevant if it “ends” with ⊥. More precisely, (Ir)relevant formulas are defined inductively by the clauses
- ⊥ is relevant, all other atomic formulas are irrelevant,
- if C is (ir)relevant and B is arbitrary, then B → C is (ir)relevant,
- if, C₀ and C₁ are (ir)relevant, then C₀ ∧ C₁ is (ir)relevant.
- if C is (ir)relevant, then ∀xC is (ir)relevant.

Definition ( Definite and goal formulas )

\[ D := P \mid G \rightarrow D \text{ (provided } D \text{ not rel. } \Rightarrow G \text{ irrel.)} \mid D \land D \mid \forall x D. \]
\[ G := P \mid D \rightarrow G \text{ (prov. } D \text{ not rel. } \Rightarrow D \text{ dec.)} \mid G \land G \mid \forall x G \text{ (prov. } G \text{ irr.)} \]
Extracted program from classical proof $\forall v \exists^c w \text{Rev}(v, w)$

Reverse :=

$(\lambda (v0) \hspace{1cm} (((\text{listrec} (\lambda (v1) \hspace{1cm} v1)) \hspace{1cm} (\lambda (n1) \hspace{1cm} (\lambda (v2) \hspace{1cm} (\lambda (f3) \hspace{1cm} (\lambda (v4) \hspace{1cm} (f3 (n1::v4)))))))))$

$\text{v0}$

$|\text{Nil}|))$

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

\[
\begin{align*}
\text{reverse } \text{Nil } v_1 & = v_1 \\
\text{reverse } (n_1 :: v_2) v_4 & = \text{reverse } v_2(n_1 :: v_4)
\end{align*}
\]

Linear instead of quadratic program!
Program extraction from classical proofs

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

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

**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.

$$\forall u, \forall^{nc} v. \ v \vdash u = v_0 \rightarrow \forall w \neg \text{Rev}(v, w).$$
The use of the (non-computational) nc-quantifier in the proof is essential!

Otherwise we would extract:

\[
\text{Reverse}_2 \nu_0 = \text{reverse}_2 \nu_0 \text{Nil Nil}
\]

\[
\begin{align*}
\text{reverse}_2 & \quad \text{Nil} \ \nu_1 \ \nu_2 = \nu_2 \\
\text{reverse}_2 & \quad (\text{Cons} \ n_1 \ \nu_2) \ \nu_4 \ \nu_5 = \text{reverse}_2 \ \nu_2 \\
& \quad \nu_4 +\!+ [n_1] \\
& \quad (n_1 :: \nu_5)
\end{align*}
\]
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.,

\[
\forall (a_i)_{i<\omega} \exists i, j. \ i < j \rightarrow a_i \leq_A a_j
\]

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:

\[
\text{DC}' \quad B([]) \land \forall s (B(s) \rightarrow \exists n B(ns * n)) \rightarrow \exists g \forall k B(\bar{g}k)
\]

DC' follows from the more common scheme of DC:

\[
\forall n \forall x^\rho \exists y^\rho A_n(x, y) \rightarrow \exists f^{\text{nat} \rightarrow \rho}. f(0) = x_0^\rho \land \forall n A_n(f(n), f(n + 1))
\]
Why are choice principles problematic for A-translation?

- $D^{X}$ is not an instance of $D^{C}$ anymore [in contrast to induction axioms, for example]
- $D^{C}$ 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 $\Delta$ and a term $t$ such that

$$\Delta \vdash t \text{ mr } A^\times.$$

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

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

Then, there is a program $p$ such that

$$\Delta \vdash D \rightarrow G[y/p].$$
Example: The infinite pigeon hole principle

**Thm**

Every boolean valued sequence \( f : \text{nat} \to \text{boole} \) 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 \( f : \text{nat} \to \text{boole} \) 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**

\[
\begin{align*}
 f & = T \ F \ T \ T \ F \ T \ T \ F \ F \ \ldots \\
n & = 4 \\
 ns = [n_0, \ldots, n_3] & = [0, 2, 3, 6]
\end{align*}
\]
A proof of the infinite pigeon principle using dependent choice.

**Thm**

*Every boolean valued sequence* $f^{\text{nat}\to\text{boole}}$ *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. $	ext{FTFFTF} \ldots \text{TTTTTTTTTTTTTT}$
   The sequence $f$ becomes constant $T$ from a certain point on, then we have a subsequence which is always $T$.

2. $	ext{FTFFFTTTTFFFTTTTTT} F \text{ TT F T} \ldots$
   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:

$$\forall f : \text{nat} \to \text{boole} \forall n \exists s : \text{list nat}. |s| = n \land \text{Inc } s \land \text{Cst } f s.$$  

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

We used the following instance of dependent choice:

$$\text{DC}' \quad B([]) \land \forall s (B(s) \to \exists n B(s \cdot n)) \to \exists g \forall k B(\bar{g} k)$$

with $B(s) := \neg \neg (\text{Inc } s \land \text{Cst } f s)$.

$\cdot$ denotes the cons operation on lists.
Realizer for DC’

**Theorem**

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

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

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.)
- $\pi_0/\pi_1$ are used for projections.

Then

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

where (DC’) is used with a relevant formula $B$. 
Realizer for DC’: Axioms needed

Let \( \# \) denote concatenation, \( \ast \) concatenation of one elt.

1. **Modified bar recursion at type \( \rho \) with the defining equation:**

   \[
   \Psi(Y, G, s) \overset{=} \equiv Y(s \# \lambda k. G(k, s, \lambda x. \Psi(Y, G, s \ast x))).
   \]

2. **Principle of continuity:**

   \[
   \forall F: (\text{nat} \to \rho) \to \tau, \alpha: \text{nat} \to \rho \exists n \forall \beta (\overline{\alpha}n = \overline{\beta}n \to F(\alpha) = F(\beta)).
   \]

3. **Principle of relativised quantifier free bar induction**

   \[
   \forall \alpha \in S \exists n P(\overline{\alpha}n) \\
   \forall s \in S. \forall x (S(s \ast x) \to P(s \ast x)) \to P(s) \\
   \frac{S([])}{P([])}
   \]
(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)=>tsil nat)_3| n1) ((fbar e2) n1)))))
  |Lin|)))
Extracted term in Minlog: $n = 3$ and general case

$n = 3$:

$f = TTTTTTTTTTT...$

$ns = [0,1,2]$

$f = FFFFFFFFFFFFFF...$

$ns = [2,5,8]$

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?

\[
\begin{align*}
F & \quad TTT \ldots \\
0 & \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11 \quad 12 \quad 13 \quad 14 \quad 15 \quad 16 \quad 17 \quad 18 \quad 19 \quad 20 \ldots
\end{align*}
\]

\( ns = [1, 2, 3, 4, 5] \)

\[
\begin{align*}
F & \quad TTTT \quad F \quad T \ldots \\
0 & \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11 \quad 12 \quad 13 \quad 14 \quad 15 \quad 16 \quad 17 \quad 18 \quad 19 \quad 20 \ldots
\end{align*}
\]

\( ns = [11, 12, 13, 14, 15] \)

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

\[
l_{11} \land l_{21} \land \{\neg l_{11} \lor \neg l_{21}\}
\]

**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:

\[
\begin{align*}
\frac{\Gamma, l \vdash \Delta}{\Gamma \vdash \Delta, \{l\}} & \quad (Unit) \\
\frac{\Gamma, l \vdash \Delta, C}{\Gamma, l \vdash \Delta, (\overline{l}, C)} & \quad (Red) \\
\frac{\Gamma, l \vdash \Delta, (l, C)}{\Gamma, l \vdash \Delta} & \quad (Elim) \\
\frac{\Gamma \vdash \Delta, \emptyset}{\Gamma \vdash \Delta} & \quad (Conflict) \\
\frac{\Gamma, l \vdash \Delta}{\Gamma, \overline{l} \vdash \Delta} & \quad (Split)
\end{align*}
\]

Here \( \Gamma \) is a valuation (set of literals, with values already assigned) and \( \Delta \) is a formula (clause set).
Valuations and Models

- A **valuation** $\Gamma$, i.e., set of literals $\{l_1, \ldots, l_k\}$ is **consistent** iff $l \in \Gamma \rightarrow \overline{l} \notin \Gamma$. Let $\text{Cons}$ be the set of all consistent Valuations.

- A **model** is a total function $M$ which maps literals to booleans and satisfies the following property $\forall l$, $M. M l \leftrightarrow \neg (M \overline{l})$

Two abbreviations:

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

We call a valuation $\Gamma$ and a formula $\Delta$ **compatible** if there exists a model satisfying both, i.e.

$$\exists M. M \models \Gamma \land M \models \Delta,$$

i.e.

$$\exists M. M \models \Gamma \land \forall C \in \Delta \exists l \in C M l.$$
Formalising and Proving Completeness

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

\[ \text{incompatible}(\Gamma; \Delta) \rightarrow \Gamma \vdash \Delta \]

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

\[ \text{compatible}(\Gamma; \Delta) \lor \Gamma \vdash \Delta \]
Formalising and Proving Completeness

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

\[
\text{incompatible}(\Gamma; \Delta) \rightarrow \Gamma \vdash \Delta
\]

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

\[
\text{compatible}(\Gamma; \Delta) \lor \Gamma \vdash \Delta
\]

Program extraction yields a program that either yields a model if \( \Gamma \) and \( \Delta \) are compatible (\( \exists M. M \models \Gamma \land M \models \Delta \)) or a derivation if incompatible.
Proof of Completeness Theorem

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

$$(\Gamma \vdash \Delta \cup \Theta) \lor \exists M. M \models \Gamma \land M \models \Delta \cup \Theta,$$

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.
We run the extracted solver using *pigeon hole formulae*

\[
\text{PHP}(n, m) := \{l_{i,1} \lor \ldots \lor l_{i,m} | 1 \leq i \leq n\} \\
\cup \{\neg l_{i,k} \lor \neg l_{j,k} | 1 \leq i < j \leq n, 1 \leq k \leq m\}
\]

Intuitively, \text{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:

<table>
<thead>
<tr>
<th>$PHP(2, 2)$</th>
<th>$PHP(3, 3)$</th>
<th>$PHP(4, 4)$</th>
<th>$PHP(5, 5)$</th>
<th>$PHP(6, 6)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 1 Sec</td>
<td>&lt; 1 Sec</td>
<td>5.45</td>
<td>26.09</td>
<td>1:34.11</td>
</tr>
</tbody>
</table>

On unsatisfiable formulae:

<table>
<thead>
<tr>
<th>$PHP(2, 1)$</th>
<th>$PHP(3, 2)$</th>
<th>$PHP(4, 3)$</th>
<th>$PHP(5, 4)$</th>
<th>$PHP(6, 5)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 1 Sec</td>
<td>1.17</td>
<td>33.62</td>
<td>13:54</td>
<td>5:35:41</td>
</tr>
</tbody>
</table>
### Extraction to Haskell:

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

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

<table>
<thead>
<tr>
<th>Formula</th>
<th>$\forall^{nc}$ compiled (Yes/No)</th>
<th>Versat</th>
</tr>
</thead>
<tbody>
<tr>
<td>PHP(7,6)</td>
<td>0.226s</td>
<td>0.089s</td>
</tr>
<tr>
<td>PHP(8,7)</td>
<td>2.42s</td>
<td>0.794s</td>
</tr>
<tr>
<td>PHP(9,8)</td>
<td>32.062s</td>
<td>17.217s</td>
</tr>
<tr>
<td>PHP(10,9)</td>
<td>15m 5s</td>
<td>15m 46s</td>
</tr>
</tbody>
</table>
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).

<table>
<thead>
<tr>
<th>Formula</th>
<th>$\forall^{nc}$ compiled</th>
<th>SCADE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Yes/No</td>
<td>Witness</td>
</tr>
<tr>
<td>SC1</td>
<td>8s</td>
<td>12s</td>
</tr>
</tbody>
</table>
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 (→ Inductive and coinductiveDefs, 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.


