Extraction of Programs from Proofs using Postulated Axioms

Anton Setzer
Swansea University, Swansea UK
(Joint work with Chi Ming Chuang)
Talk given at JAIST, Japan

22 January 2015
1. A short introduction into Agda

2. Real Number Computations in Agda

3. Theory of Program Extraction

4. Reduction of Nested to Simple Pattern Matching

5. Extensions

6. Applications

Conclusion
1. A short introduction into Agda

2. Real Number Computations in Agda

3. Theory of Program Extraction

4. Reduction of Nested to Simple Pattern Matching

5. Extensions

6. Applications

Conclusion
Agda

- Agda is a theorem prover based on Martin-Löf’s intuitionistic type theory.
- Proofs and programs are treated the same:
  \[ n : \mathbb{N} \]
  \[ n = \exp 5 20 \]
  \[ p : A \land B \]
  \[ p = \langle \cdots, \cdots \rangle \]
- Programs and proofs are defined recursively.
- In order to obtain soundness, elements of proofs need to be terminating. Otherwise we could prove falsity:
  \[ p : \bot \]
  \[ p = p \]

Termination of programs guaranteed by a termination checker based on strongly extended primitive recursion.
1. A short introduction into Agda

Framework of Agda

- For historic reasons types denoted by keyword Set.
- 3 main constructs:
  - dependent function types,
  - algebraic data types,
  - coalgebraic data types.
Dependent Function Types and ∀-Quantifier

- Dependent function type

\[(x : A) \rightarrow B\]

is type of functions mapping \(a : A\) to an element of type \(B[x := a]\).

- E.g.

\[
\begin{align*}
\text{matmult} : (n m k : \mathbb{N}) & \rightarrow \text{Mat } n m \rightarrow \text{Mat } m k \rightarrow \text{Mat } n k \\
\text{matmult} n m k A B & = \cdots
\end{align*}
\]

- Main example of dependent function type is ∀-quantifier:

\[(x : A) \rightarrow \varphi\]

is type of functions mapping \(x : A\) to a proof of \(\varphi\),
i.e. type of proofs of \(\forall x.\varphi\).
So \((x : A) \rightarrow \varphi\) stands for \(\forall x.\varphi\).
Algebraic data types

\[
data \mathbb{N} : \text{Set}\\
0 : \mathbb{N} \\
\text{suc} : \mathbb{N} \rightarrow \mathbb{N}
\]

Functions defined by pattern matching

\[
f : \mathbb{N} \rightarrow \mathbb{N}\\
f \ 0 = 5 \\
f \ (\text{suc} \ 0) = 12 \\
f \ (\text{suc} \ (\text{suc} \ n)) = (f \ n) \ast n
\]
Equality

Equality type is algebraic type indexed over pairs of elements of set $A$. There is on proof $\text{refl} : x == x$.

```agda
data _==_ \{X : Set\} : X \to X \to Set where
  refl : \{x : X\} \to x == x

transferEq : (X : Set)
  \to (Y : X \to Set)
  \to (x : X)
  \to (y : X)
  \to (x == y)
  \to Y x
  \to Y y

transferEq X Y x x refl y = y
```

Anton Setzer  
Postulated axioms in program extraction
## Coalgebraic data types

Syntax as AS would like it to be:

```agda
coalg Stream : Set where
  head : Stream → ℕ
  tail : Stream → Stream

inc : ℕ → Stream
head (inc n) = n
tail (inc n) = inc (n + 1)
```
1. A short introduction into Agda

Syntax in Agda

- Agda allows hidden arguments

\[
\text{cons} : \{ X : \text{Set} \} \rightarrow X \rightarrow \text{List } X \rightarrow \text{List } X
\]

\[
l : \text{List } \mathbb{N}
\]

\[
l = \text{cons } 0 \text{ nil}
\]

No deep theory behind – anything is legal as long as the theorem prover can determine a unique solution to hidden arguments.

- Agda has mixfix symbols.

Syntax example `if_then_else_`

Again: anything is allowed as long as the parser can parse it uniquely.

- Postulated functions (functions without a definition)

\[
\text{postulate } \text{false} : \bot
\]
Dependent Product

One example of an algebraic data type:

\[
data \exists (A : \text{Set}) (\varphi : A \rightarrow \text{Set}) : \text{Set} \\
\langle -, - \rangle : (a : A) \rightarrow \varphi a \rightarrow \exists A \varphi
\]

Projections

\[
\pi_0 : \{A : \text{Set}\} \rightarrow \{\varphi : A \rightarrow \text{Set}\} \rightarrow \exists A \varphi \rightarrow A \\
\pi_0 \langle a, b \rangle = a
\]

\[
\pi_1 : \{A : \text{Set}\} \rightarrow \{\varphi : A \rightarrow \text{Set}\} \rightarrow (x : \exists A \varphi) \rightarrow \varphi (\pi_0 x) \\
\pi_1 \langle a, b \rangle = b
\]
1. A short introduction into Agda

2. Real Number Computations in Agda

3. Theory of Program Extraction

4. Reduction of Nested to Simple Pattern Matching

5. Extensions

6. Applications

Conclusion
Question by Ulrich Berger

- Can you extract programs from proofs in Agda?
- Obvious because of Axiom of Choice?
  From
  \[ p : (x : A) \to \exists B \varphi \]
  we get of course
  \[ f = \lambda x.\pi_0 (p x) : A \to B \]
  \[ q = \lambda x.\pi_1 (p x) : (x : A) \to \varphi (f x) \]
- However what happens in the presence of axioms?
Real Numbers as Ideal Objects

- Situation different in presence of axioms.
- Approach of Ulrich Berger transferred to Agda: Axiomatice the real numbers abstractly. E.g.

```agda
postulate \( \mathbb{R} \) : Set
postulate _+_: \( \mathbb{R} \rightarrow \mathbb{R} \rightarrow \mathbb{R} \)
postulate commutative : (\( r, s : \mathbb{R} \)) \( \rightarrow r + s \equiv s + r \)
```

...
Computational Numbers as Concrete Objects

- Formulate \( \mathbb{N}, \mathbb{Z}, \mathbb{Q} \) as usual

```agda
data \( \mathbb{N} \) : Set where
  0 : \( \mathbb{N} \)
  suc : \( \mathbb{N} \rightarrow \mathbb{N} \)

_ + _ : \( \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \)
\( n + 0 = n \)
\( n + \text{suc } m = \text{suc } (n + m) \)

_ * _ : \( \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \)
...

```

```agda
data \( \mathbb{Z} \) : Set where
  ...

```

```agda
data \( \mathbb{Q} \) : Set where
  ...
```
Embedding of $\mathbb{N}, \mathbb{Z}, \mathbb{Q}$ into $\mathbb{R}$

\[
\begin{align*}
N2R : \mathbb{N} &\rightarrow \mathbb{R} \\
N2R \ 0 &\ = \ 0_R \\
N2R \ (\text{suc } n) &\ = \ N2R \ n +_R 1_R \\

Z2R : \mathbb{Z} &\rightarrow \mathbb{R} \\
\ldots \\

Q2R : \mathbb{Q} &\rightarrow \mathbb{R} \\
\ldots
\end{align*}
\]

- We obtain a link between computational types $\mathbb{N}, \mathbb{Z}, \mathbb{Q}$ and the postulated type $\mathbb{R}$. 
data CauchyReal \( r : \mathbb{R} \) : Set where
cauchyReal : \( f : \mathbb{N} \rightarrow \mathbb{Q} \)
\( p : \big( n : \mathbb{N} \big) \rightarrow |\mathbb{Q}^2_\mathbb{R} ( f \, n ) - \mathbb{R} \, r |\mathbb{R} < \mathbb{R} \, 2^{-n} \)
\( \rightarrow \text{CauchyReal} \, r \)
Show CauchyReal closed under +, *, other operations.

\[ \text{lemma : } (r \ s : \mathbb{R}) \rightarrow \text{CauchyReal } r \rightarrow \text{CauchyReal } s \rightarrow \text{CauchyReal } (r \ast s) \]

Using this show \( p : \text{CauchyReal } r \) for some \( r \).
  
  E.g. for \( r = \mathbb{Q}_2 \mathbb{R} q \).

Define

\[ f : (r : \mathbb{R}) \rightarrow (p : \text{CauchyReal } r) \rightarrow \mathbb{N} \rightarrow \mathbb{Q} \]

which extracts the Cauchy sequence in \( p \).

If we have \( r : \mathbb{R}; \ p : \text{CauchyReal } r; \ n : \mathbb{N} \) then

\[ f \ r \ p \ n : \mathbb{Q} \]

is an approximation of \( r \) up to \( 2^{-n} \). Can be computed in Agda.
Problem of Program Extraction

- Problem is that definition of \( f \) was referring to postulated axioms.
- So we might obtain

\[
 f \ r \ p \ n = \text{lemma35} \ (\text{lemma16} \ 3) \ 5
\]

- We want that even though we use postulated axioms \( f \ r \ p \ n \) reduces to a computational real number, i.e. \((1/2)\).
Signed Digit Representations

- We can consider as well the real numbers with signed digit representations.
- Signed digit representable real numbers in $[-1, 1]$ are of the form

$$0.111(-1)0(-1)01(-1)\cdots$$

In general

$$0.d_0d_1d_2d_3\cdots$$

where $d_i \in \{-1, 0, 1\}$.

- Signed digit needed because even the first digit of an unsigned digit representation can in general not be determined.
Coalgebraic Definition of Signed Digit Real Numbers (SD)

\[
\text{data Digit : Set where} \\
\quad -1_d \ 0_d \ 1_d : \text{Digit}
\]

\[
\text{coalg SD : } \mathbb{R} \rightarrow \text{Set where} \\
\quad \in [-1, 1] : \{ r : \mathbb{R} \} \rightarrow \text{SD } r \rightarrow r \in \mathbb{R} [-1, 1] \\
\quad \text{digit} : \{ r : \mathbb{R} \} \rightarrow \text{SD } r \rightarrow \text{Digit} \\
\quad \text{tail} : \{ r : \mathbb{R} \} \rightarrow (p : \text{SD } r) \rightarrow \text{SD } (2 \mathbb{R} \ast \mathbb{R} r - \mathbb{R} (\text{digit } p))
\]
Proof of \(1_R = 0.1_d 1_d 1_d 1_d \cdots\)

\[
1_{SD} : (r : \mathbb{R}) \rightarrow (r =_{\mathbb{R}} 1_R) \rightarrow SD \ r
\]

\(\in [-1, 1]\) \(1_{SD} \ r \ q \) \(= \cdots\)

digit \(1_{SD} \ r \ q \) \(= 1_d\)

tail \(1_{SD} \ r \ q \) \(= 1_{SD} (2_R *_R r -_R 1_R) \cdots\)

Proofs of \(\cdots\) can be
- inferred purely logically from axioms about \(\mathbb{R}\) (using automated theorem proving?)
- added as postulated axioms.
Proof of \(0_{\mathbb{R}} = 0.(-1_{d})1_{d}1_{d}1_{d} \cdots\)

\[
0_{SD} : (r : \mathbb{R}) \rightarrow (r =_{\mathbb{R}} 0_{\mathbb{R}}) \rightarrow SD \ r \\
\in[-1, 1] \quad (0_{SD} \ r \ q) = \cdots \\
\text{digit} \quad (0_{SD} \ r \ q) = -1_{d} \\
\text{tail} \quad (0_{SD} \ r \ q)) = 1_{SD} \ (2_{\mathbb{R}} *_{\mathbb{R}} r -_{\mathbb{R}} (-1_{\mathbb{R}})) \cdots
\]
Extraction of Programs

- From
  \[ p : \text{SD } r \]
  one can extract the first \( n \) digits of \( r \).
- Show e.g. closure of \( \text{SD} \) under \( \mathbb{Q} \cap [-1, 1], + \cap [-1, 1], \ast, \pi_{10} \cdots \)
- Then we extract the first \( n \) digits of any real number formed using these operations.
- Has been done (excluding \( \pi_{10} \)) in Agda.
First 1000 Digits of $\frac{29}{37} \times \frac{29}{3998}$
3. Theory of Program Extraction

1. A short introduction into Agda

2. Real Number Computations in Agda

3. Theory of Program Extraction

4. Reduction of Nested to Simple Pattern Matching

5. Extensions

6. Applications

Conclusion
Problem with Program Extraction

- Because of postulates it is not guaranteed that each program reduces to canonical head normal form.
- Example 1

\[
\text{postulate } \text{ax} : (x : A) \rightarrow B[x] \lor C[x]
\]

\[
a : A
\]
\[
a = \cdots
\]

\[
f : B[a] \lor C[a] \rightarrow \mathbb{B}
\]
\[
f (\text{inl } x) = \text{tt}
\]
\[
f (\text{inr } x) = \text{ff}
\]

\[
f (\text{ax } a) \text{ in Normal form, doesn’t start with a constructor}
\]

- Axioms with computational content should not be allowed.
Example 2

postulate ax : $A \land B$

$f : A \rightarrow B \rightarrow B$
$f a b = \cdots$

$g : A \land B \rightarrow B$
$g (p a b) = f a b$

$g$ ax in normal form doesn’t start with a constructor

- Problem actually occurred.
- Axioms with result type algebraic data types are not allowed.
Example 3

\[ r0 : \mathbb{R} \]
\[ r0 = 1_{\mathbb{R}} \]

\[ r1 : \mathbb{R} \]
\[ r1 = 1_{\mathbb{R}} +_{\mathbb{R}} 0_{\mathbb{R}} \]

postulate ax : \( r0 = r1 \)
postulate ax : $r_0 == r_1$

transfer : $(r \; s : \mathbb{R}) \rightarrow r == s \rightarrow \text{SD} \; r \rightarrow \text{SD} \; s$
transfer $r \; r \; \text{refl} \; p = p$

$f : (r : \mathbb{R}) \rightarrow \text{SD} \; r \rightarrow \text{Digit}$
$f \; r \; a = \cdots$

$p : \text{SD} \; r_0$
$p = \cdots$

$q : \text{SD} \; r_1$
$q = \text{transfer} \; r_0 \; r_1 \; \text{ax} \; p$

$q' : \text{Digit}$
$q' = f \; r_1 \; q$

NF of $q'$ doesn’t start with a constructor

Problem actually occurred.
Instead of defining

\[ p : \text{SD} \ r_0 \]

define

\[ p : (r : \mathbb{R}) \rightarrow (r == r_0) \rightarrow \text{SD} \ r \]
3. Theory of Program Extraction

Conditions for Correctness

- We will define conditions which guarantee that every closed term in normal form which is an element of an algebraic data type is in **canonical normal form** (starts with a constructor).
3. Theory of Program Extraction

General Assumptions about Agda Code

- Agda code is **strongly normalising**.
- Agda code is **confluent**.
- No occurrence of **record types, let- and where-expressions**.
- Apart from the identity type, all **algebraic data types** are **non-indexed** and we have **no inductive-recursive definitions**.
- **No coalgebraic types** (work in progress to include them).
- Functions defined in Agda by pattern matching have
  - a **coverage complete pattern matching** (all cases provided)
  - all **patterns** are **disjoint**.
General Assumptions about Agda Code

- Agda code is **consistent**, i.e.:
  - If Agda proves $A = B : \text{Set}$ then
    - if one is algebraic data type the other one is algebraic data type with same definition (up to equality)
    - if one is of the form $(x : B) \rightarrow C$ so is the other with equal types
  - If $t : C \ t_1 \cdots t_n : B$ where $B$ is algebraic, then $C$ is a constructor of $B$ and $t_i$ are of appropriate types.
  - If $C \ t_1 \cdots t_n = C' \ t_1 \cdots t'_m$ then $C = C'$, $n = m$, $t_i = t'_i$. 
Main Restriction on Agda Code

- If $A$ is a postulated constant then either
  - $A : (x_1 : B_1) \rightarrow \cdots \rightarrow (x_n : B_n) \rightarrow \text{Set}$ or
  - $A : (x_1 : B_1) \rightarrow \cdots \rightarrow (x_n : B_n) \rightarrow A' \; t_1 \cdots t_n$ where $A'$ is a postulated constant or an equality.

- The same applies to functions $f$ defined by case distinction on equalities.
Main Theorem

Theorem (Main Theorem)

- Assume the above conditions.
- Then every closed term in normal form which is an element of an algebraic data type is in **canonical normal form** (starts with a constructor).
Assume $t : A$, $t$ closed in normal form, $A$ algebraic data type.

Show by induction on $\text{length}(t)$ that $t$ starts with a constructor:

We have

$$t = f \, t_1 \ldots t_n$$

where $f$ function symbol or constructor.

$f$ cannot be postulated or directly defined.

$f$ cannot be defined by case distinction on an equality.

If $f$ is defined by pattern matching on an algebraic data type say $t_i$.

By IH $t_i$ starts with a constructor.

$t$ has a reduction, wasn’t in NF.

So $f$ is a constructor.
Properties of Agda Code

- Agda code has the **normal form property** if every closed normal term which is an element of an algebraic data type starts with a constructor.
- Agda code $\mathcal{A}'$ **extends** Agda code $\mathcal{A}$ ($\mathcal{A} \subseteq \mathcal{A}'$) if all judgements derivable in $\mathcal{A}$ are derivable in $\mathcal{A}'$ as well.
- Assume $\mathcal{A} \subseteq \mathcal{A}'$. 
  $\mathcal{A}'$ **induces the head normal form property on** $\mathcal{A}$ if
  - whenever $B$ is an algebraic data type
  - s.t. $\mathcal{A} \vdash t : B$
  - and $t$ has in $\mathcal{A}'$ a normal form starting with a constructor,
  - then $t$ has in $\mathcal{A}$ a normal form starting with the same constructor.
Properties of Agda Code

Assume $\mathcal{A} \subseteq \mathcal{A}'$.

- $\mathcal{A} \subseteq \mathcal{A}'$ induces the **coverage completeness property**, iff: if $\mathcal{A}$ is coverage complete with disjoint patterns so is $\mathcal{A}'$.

- $\mathcal{A} \subseteq \mathcal{A}'$ induces the **strong normalisation property**, iff: if $\mathcal{A}$ is strongly normalising, so is $\mathcal{A}'$.

- $\mathcal{A} \subseteq \mathcal{A}'$ induces the **consistency property**, iff: if $\mathcal{A}$ is consistent, so is $\mathcal{A}'$. 
Theorem (Unnesting of Pattern Matching)

Assume \( \mathcal{A} \) is Agda code fulfilling the above restrictions. Then there exists \( \mathcal{A} \subseteq \mathcal{A}' \) s.t.

1. \( \mathcal{A}' \) has simple pattern matching only,
2. \( \mathcal{A} \subseteq \mathcal{A}' \) induces the head normal form property,
3. \( \mathcal{A} \subseteq \mathcal{A}' \) induces coverage completeness, strong normalisation and consistency properties.
Example Reduction to Simple Pattern Matching

Original code:

\[
_ - _ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}
\]

\[
m \quad - \quad 0 \quad = \quad m
\]

\[
0 \quad - \quad (\text{suc } n) \quad = \quad 0
\]

\[
(\text{suc } m) \quad - \quad (\text{suc } n) \quad = \quad m - n
\]

Make sure lines make case distinction on first argument:

\[
_ - _ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}
\]

\[
0 \quad - \quad 0 \quad = \quad 0
\]

\[
(\text{suc } n) \quad - \quad 0 \quad = \quad \text{suc } n
\]

\[
0 \quad - \quad (\text{suc } n) \quad = \quad 0
\]

\[
(\text{suc } m) \quad - \quad (\text{suc } n) \quad = \quad m - n
\]
Example Reduction to Simple Pattern Matching

\[ - - : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \]

\[
\begin{align*}
0 & \quad - \quad 0 \quad = \quad 0 \\
(suc \ n) & \quad - \quad 0 \quad = \quad suc \ n \\
0 & \quad - \quad (suc \ n) \quad = \quad 0 \\
(suc \ n) & \quad - \quad (suc \ m) \quad = \quad n - m
\end{align*}
\]

Reorder lines:

\[ - - : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \]

\[
\begin{align*}
0 & \quad - \quad 0 \quad = \quad 0 \\
0 & \quad - \quad (suc \ n) \quad = \quad 0 \\
(suc \ n) & \quad - \quad 0 \quad = \quad suc \ n \\
(suc \ n) & \quad - \quad (suc \ m) \quad = \quad n - m
\end{align*}
\]
Example Reduction to Simple Pattern Matching

Make case distinction on first argument only and delegate it to auxiliary functions $e$ and $f$:

\[
\text{mutual}
\]
\[
\begin{align*}
_0 - _ & : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
0 - m & = e m \\
(suc \ n) - m & = f n m
\end{align*}
\]

\[
e : \mathbb{N} \to \mathbb{N}
\]
\[
e 0 & = 0 \\
e (suc \ n) & = 0
\]

\[
f : \mathbb{N} \to \mathbb{N} \to \mathbb{N}
\]
\[
f n 0 & = suc n \\
f n (suc \ m) & = n - m
\]
Example 2 Reduction to Simple Pattern Matching

Original code:

\[
\begin{align*}
  f : \mathbb{N} & \rightarrow \mathbb{N} \\
  f \ 0 & = 5 \\
  f \ (\text{suc} \ 0) & = 12 \\
  f \ (\text{suc} \ (\text{suc} \ n)) & = (f \ n) \ast n
\end{align*}
\]

Reduct:

\[
\begin{align*}
  \text{mutual} \\
  f : \mathbb{N} & \rightarrow \mathbb{N} \\
  f \ 0 & = 5 \\
  f \ (\text{suc} \ n) & = g \ n
\end{align*}
\]

\[
\begin{align*}
  g : \mathbb{N} & \rightarrow \mathbb{N} \\
  g \ 0 & = 12 \\
  g \ (\text{suc} \ n) & = (f \ n) \ast n
\end{align*}
\]
4. Reduction of Nested to Simple Pattern Matching

Termination of the Reductions

- If $A$ is Agda code, $f$ a function of $A$ with pattern matching terms

$$m^A(f) := \begin{cases} 0 & \text{if } f \text{ has simple pattern matching} \\ \text{sum of length of all patterns of } f & \text{otherwise} \end{cases}$$

- Let for Agda code $A$

$$m(A) = \{ |m^A(f)| \mid f \text{ function symbol defined by pattern matching in } A \}$$

where $\{ |\cdots| \}$ denotes a multiset.
Main Difficulty

- Show that each reduction step induces the properties mentioned before.
Proof of Main Theorem

- First reduce Agda code to simple pattern matching using Theorem on Unnesting of Pattern Matching.
- Then use the above proof for Agda code having simple pattern matching.
1. A short introduction into Agda

2. Real Number Computations in Agda

3. Theory of Program Extraction

4. Reduction of Nested to Simple Pattern Matching

5. Extensions

6. Applications

Conclusion
Extensions

- Negated axioms such as \(\neg(0_R \equiv 1_R)\) are currently forbidden
  - Have form \(0_R \equiv 1_R \rightarrow \bot\) where \(\bot\) is algebraic data type.
  - Causes problems since they are needed (e.g. when using the reciprocal function).
  - Without negated axioms the theory is trivially consistent (interpret all postulate sets as one element sets).
  - With negated axioms it could be inconsistent.
    - E.g. take axioms which have consequences \(0_R \equiv 1_R\) and \(\neg(0_R \equiv 1_R)\).
  - In case of an inconsistency we would get a proof \(p : \bot\) and therefore
    \[\text{efq } p : \mathbb{N}\]
    is non-canonical of \(\mathbb{N}\) in NF.
5. Extensions

Theorem (Negated Axioms)

- Assume conditions as before.
- Assume result type of axioms is always a postulated type or a negated postulated type.
- Assume the Agda code doesn’t prove \( \bot \).
- Then every closed term which is an element of an algebraic data type is in canonical normal form (starts with a constructor).
More Extensions

- We could separate our algebraic data types into those for which we want to use their computational content and those for which we don’t use their content.
- Assume we never derive using case distinction on a non-computational data type an element of a computational data type.
- Then axioms with result type non-computational data types could be allowed, e.g.

  \[ \text{tertiumNonDatur} : A \lor_{\text{non-computational}} \neg A \]
Addition of Coalgebraic Types

- Original proof didn’t include coalgebraic types.
- With coalgebraic types additional complication: 
  \( t \) can be of the form 
  \( \text{elim } t_1 \)
  for an eliminator \( \text{elim} \) of a coalgebraic type.
- Extend the theorem by proving simultaneously:
  - If \( A \) algebraic, \( t \) closed term in NF, \( t : A \), then \( t \) starts with a constructor.
  - If \( A \) coalgebraic, \( t \) closed term, \( t : A \), and \( \text{elim} \) is an eliminator of \( A \), then \( \text{elim } t \) has a reduction.
6. Applications

1. A short introduction into Agda

2. Real Number Computations in Agda

3. Theory of Program Extraction

4. Reduction of Nested to Simple Pattern Matching

5. Extensions

6. Applications

Conclusion
Easy Proofs

- Acclimatised theory allows to easily prove big theorems by postulating them, as long as we are only interested in the computational content.
- In an experiment we introduced axioms such as

\[
\text{ax} : (r : \mathbb{R}) \rightarrow (q : \mathbb{Q}) \rightarrow |\mathbb{Q}2 \mathbb{R} q - r| <_{\mathbb{R}} 2_{\mathbb{R}}^{-2} \rightarrow q \leq_{\mathbb{Q}} 1/4_{\mathbb{Q}}
\]

\[
\rightarrow r \leq_{\mathbb{R}} 1/2_{\mathbb{R}}
\]

- In fact the more is postulated the faster the program (and the easier one can see what is computed).
Postulates allow us to have a two-layered theory with
- computational part (using non-postulated types)
- an a logic part (using postulated types).
Useful for Programming with Dependent Types

- This could be very useful for programming with dependent types.
  - Postulate axioms with no computational content.
  - Possibly prove them using automated theorem provers (approach by Bove, Dybjer et. al.).
  - Concentrate in programming on computational part.
Experiments carried out

- In about 6 hours I developed a framework using Cauchy Reals, Signed Digit Reals, conversion into streams and lists form scratch.
  - Allowed the computation of the first 10 digits of rational numbers in $[-1, 1]$.
- Framework is easy to use since most proofs are replaced by postulates.
- Chi Ming Chuang showed closure of signed digit reals under average and multiplication using more efficient direct calculations and full proofs of most theorems needed.
- Was able to calculated fast the first 1000 digits of rational numbers.
6. Applications

Idea: Type Theory with Partial and Total Objects

▶ One could postulate
  ▶ types of partial elements,
  ▶ constants operating on those types,
  ▶ equations for those constants .

▶ Then one can
  ▶ define predicates on those partial elements corresponding to the total elements,
  ▶ and show that certain partial elements are total or have other properties.
Example

postulate \( \mathbb{N}_{\text{partial}} : \) Set
postulate \( - == - : \mathbb{N}_{\text{partial}} \to \mathbb{N}_{\text{partial}} \to \text{Set} \)
postulate \( 0 : \mathbb{N}_{\text{partial}} \)
postulate \( \text{suc} : \mathbb{N}_{\text{partial}} \to \mathbb{N}_{\text{partial}} \)
postulate \( f : \mathbb{N}_{\text{partial}} \to \mathbb{N}_{\text{partial}} \)
postulate \( \text{lemf0} : f \ 0 == \ldots \)
postulate \( \text{lemfs} : (n : \mathbb{N}_{\text{partial}}) \to f \ (\text{suc} \ n) == \ldots \)
data \( \mathbb{N} : \mathbb{N}_{\text{partial}} \to \text{Set} \) where
  zero : \( \mathbb{N} \ 0 \)
  succp : \( (n : \mathbb{N}_{\text{partial}}) \to \mathbb{N} \ n \to \mathbb{N} \ (\text{suc} \ n) \)
  eqp : \( (n \ m : \mathbb{N}_{\text{partial}}) \to \mathbb{N} \ n \to n == m \to \mathbb{N} \ m \)

lemma : \( (n : \mathbb{N}_{\text{partial}}) \to \mathbb{N} \ n \to \mathbb{N} \ (f \ n) \)
lemma \( n \ p = \ldots \)
1. A short introduction into Agda

2. Real Number Computations in Agda

3. Theory of Program Extraction

4. Reduction of Nested to Simple Pattern Matching

5. Extensions

6. Applications

Conclusion
If result types of postulated constants are postulated types, then closed elements of algebraic types evaluate to constructor normal form.

Reduces the need burden of proofs while programming (by postulating axioms or proving them using ATP).

Axiomatic treatment of $\mathbb{R}$.

Program extraction for proofs with real number computations works very well.

Applications to programming with dependent types in general.

Possible solution for type theory with partiality and totality.