Auto in Agda

Wen Kokke

June 12, 2015

Programming proof search using reflection

Proofs and programs

- In a language with dependent types, “programs are proofs” and “types are propositions”.
- Proof terms can be brittle and are often tedious to write.

Evenness

```
data Even : ℕ → Set where
  base : Even 0
  step : ∀ {n} → Even n → Even (2 + n)

even8 : Even 8
even8 = step (step (step (step base)))
```

Evenness

```
data Even : ℕ → Set where
  base  :  Even 0
  step  :  ∀ {n} → Even n → Even (2 + n)

even8 : Even 8
even8 = step (step (step (step base)))
```

There is a clear need for automation…

```
even1024 : Even 1024
even1024 = ...
```

Proof by reflection

```
data ⊤ : Set where tt : ⊤
data ⊥ : Set where

even? : ℕ → Set
even?               0    = ⊤
even?               1    = ⊥
even? (suc (suc  n))  = even? n
```

Proof by reflection

```
data ⊤ : Set where tt : ⊤
data ⊥ : Set where

even? : ℕ → Set
even?            0     = ⊤
even?            1     = ⊥
even? (suc (suc  n))  = even? n

even1024 : even? 1024
even1024 = tt
```

Soundness

$$
\begin{array}{llll}
\text{soundness} : (n : \mathbb{N}) \to \text{even?}\ n \to \text{Even}\ n & & & \\
\text{soundness} & 0 & e & = \text{base} \\
\text{soundness} & 1 & () & \\
\text{soundness}\ (\text{suc}\ (\text{suc} & n)) & e & = \text{step}\ (\text{soundness}\ n\ e)
\end{array}
$$

Soundness

$$
\begin{aligned}
&\mathsf{soundness} : (n : \mathbb{N}) \to \mathsf{even?}\ n \to \mathsf{Even}\ n \\
&\mathsf{soundness}\ \quad\quad\quad\quad\quad 0 \quad\ e \ = \mathsf{base} \\
&\mathsf{soundness}\ \quad\quad\quad\quad\quad 1 \quad\ () \\
&\mathsf{soundness}\ (\mathsf{suc}\ (\mathsf{suc}\ \ n))\ \ e \ = \mathsf{step}\ (\mathsf{soundness}\ n\ e)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{even1024} : \mathsf{Even}\ 1024 \\
&\mathsf{even1024} = \mathsf{soundness}\ 1024\ \mathsf{tt}
\end{aligned}
$$

Open terms

But what to do for open terms?

$$\mathsf{lemma} : \forall\ \{n\} \rightarrow \mathsf{Even}\ n \rightarrow \mathsf{Even}\ (n + 1024)$$
$$\mathsf{lemma} = \ldots$$

Open terms

But what to do for open terms?

$$\mathsf{lemma} : \forall\ \{n\} \rightarrow \mathsf{Even}\ n \rightarrow \mathsf{Even}\ (n + 1024)$$
$$\mathsf{lemma} = \mathsf{auto}$$

Open terms

But what to do for open terms?

$$\mathsf{lemma} : \forall\ \{n\} \rightarrow \mathsf{Even}\ n \rightarrow \mathsf{Even}\ (n + \mathsf{1024})$$
$$\mathsf{lemma} = \mathsf{tactic}\ (\mathsf{auto}\ \mathsf{5}\ \mathsf{db})$$

How `auto` works

We

1. quote the current goal;
2. translate the Agda AST to our own term data type;
3. run proof search;
4. translate the resulting term data type to an Agda AST;
5. unquote the resulting AST.

How proof search works

We

1. start out with our goal;
2. fork and try to unify the goal with all of our rules' conclusions;
3. add premises as subgoals to the queue;
4. recurse.

If we ever run out of subgoals, we stop.

Terms and unification

```
data MyTerm : Set where
  var  : ℕ                        → MyTerm
  con  : Name → List MyTerm  → MyTerm

unify : (x y : MyTerm) → Maybe Subst
unify = ...
```

Terms and unification

```
data MyTerm (n : ℕ) : Set where
  var  : Fin n                       → MyTerm n
  con  : Name → List (MyTerm n)  → MyTerm n

unify : ∀ {n} (x y : MyTerm n) →  Maybe (∃ (Subst n))
unify = ...
```

Inference rules

```
record Rule (n : ℕ) : Set where
  constructor rule
  field
    name       : Name
    conclusion : MyTerm n
    premises   : List (MyTerm n)

arity : ∀ {n} (r : Rule n) → ℕ
arity = length ∘ premises
```

A ‘hint database’ is a list of rules.

Proof trees

```
data SearchTree (A : Set) : Set where
  leaf  : A                            → SearchTree A
  node  : List (∞ (SearchTree A))  → SearchTree A

fail : ∀ {A} → SearchTree A
fail = node []
```

Proofs

```
data Proof : Set where
  con : (name : Name) (args : List Proof) → Proof
```

Proofs

```
data Proof : Set where
  con : (name : Name) (args : List Proof) → Proof

PartialProof : ℕ → Set
PartialProof m =
  ∃ (λ k → Vec (MyTerm m) k × (Vec Proof k → Proof))

app : ∀ {n k}
  → (r : Rule n)
  → Vec Proof (arity r + k)
  → Vec Proof (suc k)
```

Building the search tree

We can build up a lazy SearchTree using backward-chaining search:

$$\begin{aligned}&\mathsf{solve}\\&\quad : \forall\ \{m\}\ (g : \mathsf{MyTerm}\ m) \to \mathsf{HintDB} \to \mathsf{SearchTree}\ \mathsf{Proof}\\&\mathsf{solve}\ g\ db = \ldots\end{aligned}$$

Building the search tree

We can build up a lazy SearchTree using backward-chaining search:

```
solve
  : ∀ {m} (g : MyTerm m) → HintDB → SearchTree Proof
solve g db = solveAcc (1 , g :: [] , head) db

solveAcc
  : ∀ {m} → PartialProof m → HintDB → SearchTree Proof
solveAcc {m} (     0 ,      [] ,  p) db  = leaf (p [])
solveAcc {m} (suc  k , g ::  gs ,  p) db  = node (map next db)
```

Building the search tree (cont'd)

In next, we then:

1. see if the conclusion can be unified with the current goal;
2. raise the variables in the rule by m to avoid conflict;
3. prepend the premises to the list of current goals;
4. apply the rule to the partial proof;
5. call solveAcc with the new partial proof.

Traversing the search tree

We can traverse the lazy SeachTree using, e.g. depth-first search:

$$\begin{aligned}
&\mathsf{dfs} : \forall \, \{A\} \; (\mathit{depth} : \mathbb{N}) \to \mathsf{SearchTree}\; A \to \mathsf{List}\; A \\
&\mathsf{dfs}\;\; \mathsf{zero} \qquad\quad _ \qquad\qquad\quad = [] \\
&\mathsf{dfs}\;\; (\mathsf{suc}\; k)\;\; (\mathsf{leaf}\; x) \qquad = x :: [] \\
&\mathsf{dfs}\;\; (\mathsf{suc}\; k)\;\; (\mathsf{node}\; xs) \;\; = \mathsf{concatMap}\; (\lambda\; x \to \mathsf{dfs}\; k\; (\flat\; x))\; xs
\end{aligned}$$

Where $\flat$ is Agda's notation for 'force'.

Missing pieces

We

1. quote the current goal;
2. translate the Agda AST to our own term data type;
3. **run proof search**;
4. translate the resulting term data type to an Agda AST;
5. unquote the resulting AST.

Connecting to Agda's reflection

```
idTerm : Term
idTerm = quoteTerm (λ {A : Set} (x : A) → x)
```

Connecting to Agda's reflection

```
idTerm : Term
idTerm = quoteTerm (λ {A : Set} (x : A) → x)

idTest : idTerm ≡ lam hidden (lam visible (var 0 []))
idTest = refl
```

Connecting to Agda's reflection

$$\mathsf{idTerm} : \mathsf{Term}$$
$$\mathsf{idTerm} = \mathsf{quoteTerm}\ (\lambda\ \{A : \mathsf{Set}\}\ (x : A) \to x)$$

Connecting to Agda's reflection

```
idTerm : Term
idTerm = quoteTerm (λ {A : Set} (x : A) → x)

const : {A B : Set} → A → B → A
const = unquote (lam visible (lam visible (var 1 [])))
```

Connecting to Agda's reflection

```
idTerm : Term
idTerm = quoteTerm (λ {A : Set} (x : A) → x)

const : {A B : Set} → A → B → A
const = unquote (lam visible (lam visible (var 1 [])))

lemma : ∀ {n} → Even n → Even (n + 1024)
lemma = quoteGoal g in ...
```

Why we won't talk about the translations…

```
data Term : Set where
  var           : ℕ           → List (Arg Term) → Term
  con           : Name        → List (Arg Term) → Term
  def           : Name        → List (Arg Term) → Term
  lam           : Visibility  → Term            → Term
  pat-lam       : List Clause → List (Arg Term) → Term
  pi            : Arg Type    → Type            → Term
  sort          : Sort                          → Term
  lit           : Literal                       → Term
  quote-goal    : Term                          → Term
  quote-term    : Term                          → Term
  quote-context :                                 Term
  unquote-term  : Term        → List (Arg Term) → Term
  unknown       :                                 Term
```

Overview

Assuming we have some conversions from and to Agda…

```
postulate
  fromAgda : Term  → Maybe (∃ MyTerm)
  toAgda   : Proof → Term
```

Overview

Assuming we have some conversions from and to Agda…

```
postulate
  fromAgda : Term  → Maybe (∃ MyTerm)
  toAgda   : Proof → Term
```

…the auto tactic works as follows:

```
auto : (depth : ℕ) → HintDB → Term → Term
auto depth db goal  with fromAgda goal
... | nothing  = isNotFirstOrder
... | just (m , g)     with solve g db
... | searchTree       with dfs depth searchTree
... | []         = noProofFound
... | (p :: _)   = toAgda p
```

Overview (cont'd)

Proof automation can be just like regular programming!

There are some limitations to `auto`:

- it only handles terms with first-order types;
- it's not blazingly fast.

An `auto` tactic, in general, is not very intelligent.

Tactics for natural numbers

```
data Exp (Atom : Set) : Set where
  var    : (x : Atom)        → Exp Atom
  lit    : (n : ℕ)           → Exp Atom
  _⟨+⟩_  : (e e₁ : Exp Atom) → Exp Atom
  _⟨*⟩_  : (e e₁ : Exp Atom) → Exp Atom
```

Tactics for natural numbers

```
data Exp (Atom : Set) : Set where
  var    : (x : Atom)         → Exp Atom
  lit    : (n : ℕ)            → Exp Atom
  _⟨+⟩_  : (e e₁ : Exp Atom)  → Exp Atom
  _⟨*⟩_  : (e e₁ : Exp Atom)  → Exp Atom

auto−proof : ∀ e₁ e₂ ρ → Maybe (⟦ e₁ ⟧e ρ ≡ ⟦ e₂ ⟧e ρ)
auto−proof e₁ e₂ ρ  with norm e₁ == norm e₂
auto−proof e₁ e₂ ρ  | no _        = nothing
auto−proof e₁ e₂ ρ  | yes nfeq    = ...
```

Tactics for natural numbers

```
data Exp (Atom : Set) : Set where
  var     : (x : Atom)         → Exp Atom
  lit     : (n : ℕ)            → Exp Atom
  _⟨+⟩_   : (e e₁ : Exp Atom)  → Exp Atom
  _⟨*⟩_   : (e e₁ : Exp Atom)  → Exp Atom

auto−proof : ∀ e₁ e₂ ρ → Maybe (⟦ e₁ ⟧e ρ ≡ ⟦ e₂ ⟧e ρ)
auto−proof e₁ e₂ ρ  with norm e₁ == norm e₂
auto−proof e₁ e₂ ρ  | no _        = nothing
auto−proof e₁ e₂ ρ  | yes nfeq  = ...

auto−tactic : Term → Term
auto−tactic t = ...
```

Tactics for natural numbers

```
on–goal : Name → Term
on–goal tac =
  quote–goal
    $ abs "g"
    $ unquote–term (def tac (vArg (var 0 []) :: [])) []
```

Tactics for natural numbers

```
on-goal : Name → Term
on-goal tac =
  quote-goal
    $ abs "g"
    $ unquote-term (def tac (vArg (var 0 []) :: [])) []

macro
  auto : Term
  auto = on-goal (quote auto-tactic)
```

Tactics for natural numbers

$$\text{auto-example}_1 : (a\ b : \mathbb{N}) \to (a \mathbin{\dot-} b) * (a + b) \equiv a \mathbin{\char`^} 2 \mathbin{\dot-} b \mathbin{\char`^} 2$$
$$\text{auto-example}_1\ a\ b = \text{auto}$$

$$\text{auto-example}_2 : (a\ b : \mathbb{N}) \to (a + b) \mathbin{\char`^} 2 \geq a \mathbin{\char`^} 2 + b \mathbin{\char`^} 2$$
$$\text{auto-example}_2\ a\ b = \text{auto}$$

Future Work

- macro functions can *only* take quoted arguments

Future Work

- macro functions can *only* take quoted arguments

```
data Q {a} (A : Set a) : Set where
  q : Term → Q A

macro
  plus–to–times : Q ℕ – > Q ℕ
  plus–to–times = ...

macro
  auto : (depth : ℕ) → HintDB → Term
  auto = ...
```

Conclusion

- Proof automation can be just like regular programming!
- In bleeding-edge Agda, one can implement tactics without much syntactic noise.
- An `auto` tactic can be useful for putting programs together in a robust manner; not for proof search.
- Understanding the problem space and writing a fast decision procedure is much more useful, but also takes much more effort.