Auto in Agda

joint work with Pepijn Kokke

23/5/2014
Nijmegen

What is a proof?

How can I convince my proof assistant that a proposition holds?

Proof languages

- ‘Mizar’ – domain specific language.
- ‘HOL’ – a small core theory; tactics may be built into the proof assistant.
- ‘Coq’ – proof terms generated by tactics.
- ‘Agda’ – raw proof terms.

Even

```
data Even  : ℕ →  Set where
   Base : Even 0
   Step : Even n → Even (suc (suc n))
```

Even

```
data Even  : ℕ →  Set where
    Base : Even 0
    Step : Even n → Even (suc (suc n))

even1024 : Even 1024
even1024 = …
```

There's a clear need for automation…

Type theory

Type theory is a language for *computation* and *proof*.

> *It no longer seems possible to distinguish the discipline of programming from constructive mathematics.*

Martin Löf; Constructive Mathematics and Computer Programming.

An alternative definition

```
data Empty : Set where

data True : Set where
  tt : True

even? : ℕ -> Set
even? zero           = True
even? (suc zero)     = Empty
even? (suc (suc n))  = even? n
```

An alternative definition

```
data Empty : Set where

data True : Set where
  tt : True

even? : ℕ -> Set
even? zero              = True
even? (suc zero)        = Empty
even? (suc (suc n))     = even? n

even1024 : even? 1024
even1024 = tt
```

Proof-by-reflection

```
soundness : (n : ℕ) -> even? n -> Even n
soundness zero          e    = Base
soundness (suc zero)    ()
soundness (suc (suc n)) e    = Step (soundness n e)

even1024 : Even 1024
even1024 = soundness 1024 tt
```

Even – again

```
even+ : Even n -> Even m -> Even (n + m)
even+ Base       e2 = e2
even+ (Step e1) e2 = Step (even+ e1 e2)

simple : ∀ {n} → Even n → Even (n + 1024)
simple e = …
```

Proof-by-reflection

- Works well if you have single, fixed domain:
  - parity of constants;
  - equations over a ring;
  - propositions in first-order logic…
- But what if you don't know this *a priori?*

Auto in Agda

- We’ve been thinking about how to provide *general purpose* lightweight proof automation – a bit like *auto* in Coq.
- Key ingredient: Agda’s reflection mechanism.

Reflection API

```
data Term : Set where
  -- Variable applied to arguments.
  var     : (x : ℕ) (args : List (Arg Term)) → Term
  -- Constructor applied to arguments.
  con     : (c : Name) (args : List (Arg Term)) → Term
  -- Identifier applied to arguments.
  def     : (f : Name) (args : List (Arg Term)) → Term
  -- Different kinds of λ-abstraction.
  lam     : (v : Visibility) (t : Term) → Term
  -- Pi-type.
  pi      : (t₁ : Arg Type) (t₂ : Type) → Term
  -- A sort.
  sort    : Sort → Term
  -- Anything else.
  unknown : Term
```

Demo

Proof automation

- A single function for proof automation:

```
auto : ℕ → HintDB → Term → Term
```

- Implemented in 'safe' Agda;
- Even if it may fail to produce the Term you were hoping for…

How auto works

1. Quote the current goal;
2. Translate the goal to my own Term data type;
3. Run Prolog resolution with this Term as goal;
4. Build an Agda AST from this result;
5. Unquote the AST.

Proof automation in Agda

1. Quote the current goal;
2. Translate the goal to my own Term data type;
3. **Run Prolog resolution with this Term as goal;**
4. Build an Agda AST from this result;
5. Unquote the AST.

Terms and unification

```
data Term (n : ℕ) : Set where
    var : (x : Fin n) → Term n
    con : (s : TermName) (ts : List (Term n)) → Term n

unify : (t₁ t₂ : Term m) → Maybe (Subst m)
```

Terms and unification

```
data Term (n : ℕ) : Set where
    var : (x : Fin n) → Term n
    con : (s : TermName) (ts : List (Term n)) → Term n

unify : (t₁ t₂ : Term m) → Maybe (Subst m)
unify t₁ t₂ = unifyAcc t₁ t₂ nil


unifyAcc : (t₁ t₂ : Term m) → Subst m → Maybe (Subst m)
```

Terms and unification

```
data Term (n : ℕ) : Set where
    var : (x : Fin n) → Term n
    con : (s : TermName) (ts : List (Term n)) → Term n

unify : (t₁ t₂ : Term m) → Maybe (Subst m)
unify t₁ t₂ = unifyAcc t₁ t₂ nil

unifyAcc : (t₁ t₂ : Term m) → Subst m → Maybe (Subst m)
```

(Ignoring details about number of variables)

Prolog rules

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

A ‘hint database’ is a list of rules

Prolog resolution

```
while there are open goals
  apply each rule to try to resolve the next goal
  if this succeeds
     add premises of the rule to the open goals
     continue the resolution
  otherwise fail and backtrack
```

Prolog resolution

while there are open goals

  apply each rule to try to resolve the next goal

  if this succeeds

    add premises of the rule to the open goals

    continue the resolution

  otherwise fail and backtrack

Idea: encode this (possibly) infinite process as a coinductive data type

Resolution

```
data SearchSpace (m : ℕ) : Set where
    fail : SearchSpace m
    retn : Subst m → SearchSpace m
    step : (Rule → ∞ (SearchSpace m)) → SearchSpace m

resolveAcc : Maybe (Subst m) → List (Goal m) → SearchSpace m
resolveAcc nothing       _                    = fail
resolveAcc (just subst) []                    = retn s
resolveAcc (just subst) (goal :: goals) = step next
  where
  next : Rule m → ∞ (SearchSpace m)
  next r =
     let subst′ = unifyAcc goal (conclusion r) subst in
     resolveAcc subst′ (premises r ++ goals)
```

Resolution

- It's easy to kick off the resolution process:

```
resolve : Goal m → SearchSpace m
resolve g = resolveAcc (just nil) [ g ]
```

- I'm ignoring the generation of free variables – which makes things pretty messy…

- I haven't said anything about the hint database yet.

Search trees

```
data SearchTree (A : Set) : Set where
    fail : SearchTree A
    retn : A → SearchTree A
    fork : List (∞ (SearchTree A)) → SearchTree A

toTree : Rules → SearchSpace m → SearchTree (Subst m)
toTree hints fail     = fail
toTree hints (retn s) = retn s
toTree hints (step f) = fork (map (\r -> toTree (f r)) hints)
```

(Ignoring forcing and guardedness)

Alternatives

- Apply every rule at most once;
- Assign priorities to the order in which rules may be applied;
- Limit the applications of some rules – like transitivity.
- …

Finding solutions

- We can use a simple depth-bounded search

```
dbs : (depth : ℕ) → SearchTree A → List A
```

- Or implement breadth-first search;
- Or any other traversal of the search tree.

Missing pieces

- Conversion from AgdaTerms to our Term type;
- Constructing hint databases;
- Building an AgdaTerm from a list of rules that have been applied;
- Adding error messages.

Type classes for cheap!

Reflections

- Lots of limitations:
  - first-order;
  - no information from local context;
  - slowish.
- But it works!

Conclusion

It no longer seems possible to distinguish the discipline of programming from constructive mathematics

Conclusion

It no longer seems possible to distinguish the discipline of programming from ***the construction of mathematics***