Auto in Agda

joint work with Pepijn Kokke

IFIP WG 2.1 #71
Zeegse, the Netherlands

Proofs & Programs

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

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…

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 + 2)
simple e = …
```

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)
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

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;
- Converting such a Term back to an AgdaTerm.
- Adding error messages.

Type classes for cheap!

Conclusions

- Lots of limitations:
  - first-order;
  - no information from local context;
  - slow.
- Proof automation need not be different from regular programming.