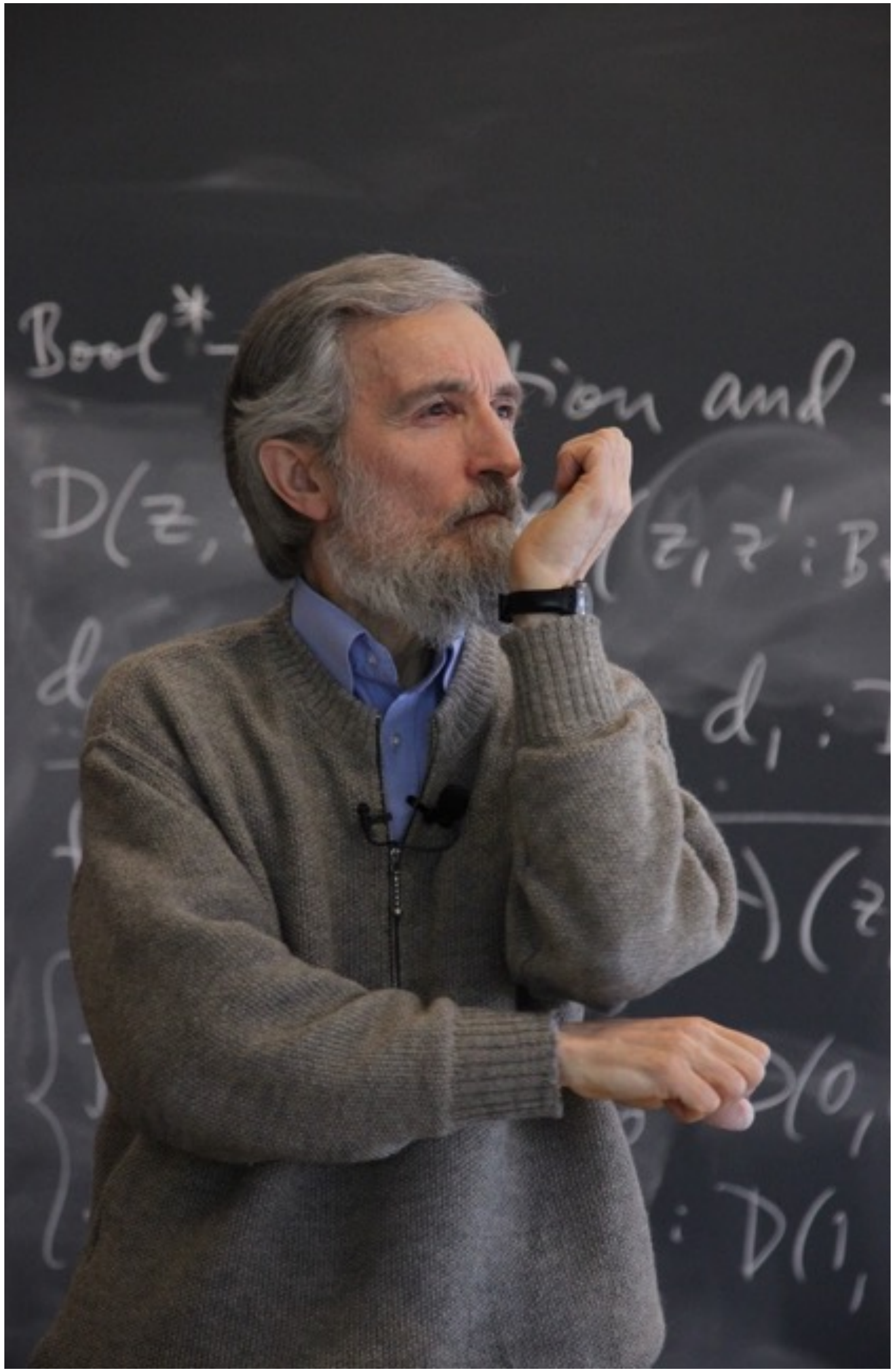
Auto in Agda

joint work with Pepijn Kokke

MPC 2015
Königswinter

Per Martin-Löf

“The intuitionistic type theory,…, may equally well be viewed as a programming language.” – Constructive Mathematics and Computer Programming ‘79



Type theory provides
a *single language*
for proofs, programs, and specs.

Coq

- Gallina – a small functional programming language
- Tactics – commands that generate proof terms
- Ltac – a tactic scripting language
- ML-plugins – add custom tactics to proof assistant

What happened to the idea of a single language?

Introducing Agda

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

even4 : Even 4
even4 = Step (Step Base)

even1024 : Even 1024
even1024 = …
```

A definition that computes

```
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
```

Proof-by-reflection

- Works very well for *closed problems,* without variables or additional hypotheses

- You can implement ‘solvers’ for a fixed domain (such as Agda’s monoid solver or ring solver), although there may be some ‘syntactic overhead’.

- But sometimes the automation you would like is more *ad-hoc*.

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 = …
```

We need to give a proof term by hand…

Maintaining hand-written proofs

- **Brittle**
- **Large**
- **Incomplete**

Even – automatic

```
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 = tactic (auto 5 hints)
```

The auto function performs proof search,
trying to prove the current goal from some list of ‘hints’

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 (4 + n)
simple = tactic (auto 5 hints)
```

Our definition is now more robust.
Reformulating the lemma does not need proof refactoring.

Use reflection
to *generate* proof terms

Agda's reflection mechanism

- A built-in type `Term`
- Quoting a term, `quoteTerm`, or goal, `quoteGoal`
- Unquoting a value of type term, splicing back the corresponding concrete syntax.

```
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 flavours of λ-abstraction.
  lam     : (v : Visibility) (t : Term) → Term
  -- Pi-type.
  pi      : (t₁ : Arg Type) (t₂ : Type) → Term
  …
```

Automation using reflection

```
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 = quoteGoal g in unquote(…g…)
```

Automation using reflection

```
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 = tactic(λ g → …g…)
```

All I need to provide here is a function from `Term` to `Term`

Examples

```
hints : HintDB
hints = [] << quote Base
           << quote Step
           << quote even+

test₁ : Even 4
test₁ = tactic (auto 5 hints)

test₂ : ∀ {n} → Even n → Even (n + 2)
test₂ = tactic (auto 5 hints)

test₃ : ∀ {n} → Even n → Even (4 + n)
test₃ = tactic (auto 5 hints)
```

How auto works

1. Quote the current goal;
2. Translate the goal to our own Term data type;
3. First-order proof search with this Term as goal;
4. Build an Agda Term from the result;
5. Unquote this final Term.

Proof automation in Agda

1. Quote the current goal;
2. Translate the goal to our own Term data type;
3. **First-order proof search 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 : Name) (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-style resolution

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

Search trees

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

Such trees are finitely branching,
but (potentially) infinitely deep.

Overview

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

Unfinished : ℕ → Set
Unfinished m = List (Goal m) × (List Proof → Proof)

search : Goal m → HintDB → SearchTree Proof

searchAcc : ∀ {m} → Unfinished m → HintDB →
   SearchTree Proof
```

The (almost) complete algorithm

```
searchAcc : ∀ {m} → Unfinished m → HintDB → SearchTree Proof
searchAcc ([] , p) _       = leaf (p [])
searchAcc (g ∷ gs , p) db = node (map step (getHints db))
  where
    step : ∃[ δ ] (Hint δ) → ∞ (SearchTree Proof)
    step (δ , h) with unify g (conclusion h)
    ... | nothing        = ♯ node [] -- fail
    ... | just (mgu)     = ♯ solveAcc uprf db
      where
        uprf : Unfinished n
        uprf = newGoals , (p ∘ con h)
          where
          prm = premises h
          newGoals  = prm ++ gs
```

Resolution

- It’s easy to kick off the resolution process with a single goal;
- I’m ignoring the generation of free variables – which makes things pretty messy…
- I’m ignoring the application of substitutions arising from unification.

Finding solutions

- We can use a simple depth-bounded search

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

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

Implementing auto

- First convert the goal to our own term type;
- if this fails, generate an error;
- otherwise, build up the search tree and traverse it using a depth-bounded search.
- if this produces at least one proof, turn it into a built-in term, ready to be unquoted.
- if this doesn't find a solution, generate an error term.

Alternatives

- Apply every rule at most once;
- Assign priorities to the rules;
- Limit when or how some rules are used.
- …

Example - sublists

```
data Sublist : List a -> List a -> Set where
  Base : ∀ ys -> Sublist [] ys
  Keep : ∀ x xs ys -> Sublist xs ys -> Sublist (x :: xs) (x :: ys)
  Drop : ∀ x xs ys -> Sublist xs ys -> Sublist xs (x :: ys)

reflexivity : ∀ xs -> Sublist xs xs

transitivity : ∀ xs ys zs ->
  Sublist xs ys -> Sublist ys zs -> Sublist xs ys

sublistHints : HintDB
```

Example – sublists

```
wrong : ∀ x -> Sublist (x :: []) []
wrong = tactic (auto 5 sublistHints)
```

What happens?

Missing from the presentation

- Conversion from Agda’s Term to our Term type;
- Building an Agda Term to unquote from a list of rules that have been applied;
- Generating rules from lemma names.
- …

Discussion

- Lots of limitations:
  - first-order;
  - limited information from local context;
  - not very fast – and it's hard to tell how to fix this!
- Constructing mathematics is indistinguishable from computer programming.

Conclusion