```
--- === Agda Tactics Programming === ---
---
---            Ulf Norell
---  wg2.8 Kefalonia, May 27, 2015

module Slides where

open import Prelude      -- https://github.com/UlfNorell/agda-prelude
open import Tactic.Nat



--- === Introduction === ---

downFrom : Nat → List Nat
downFrom  0      = []
downFrom (suc n) = suc n :: downFrom n

theorem : ∀ n → sum (map (_^ 2) (downFrom n)) * 6 ≡
                n * (n + 1) * (2 * n + 1)
theorem = induction



--- === Decision Procedures === ---

-- Basic idea:
--   - represent problem domain as a data type
--   - write a function to decide if a problem
--     is solvable
--   - prove that the function is sound

module Exp where

  infixl 6 _⟨+⟩_
  infixl 7 _⟨*⟩_
  data Exp (Atom : Set) : Set where
    var : (x : Atom) → Exp Atom
    lit : (n : Nat) → Exp Atom
    _⟨+⟩_ _⟨*⟩_ : (e e₁ : Exp Atom) → Exp Atom

  Env : Set → Set
  Env Atom = Atom → Nat

  ⟦_⟧e : ∀ {Atom} → Exp Atom → Env Atom → Nat
```

```
  ⟦ var x ⟧e   ρ = ρ x
  ⟦ lit n ⟧e   ρ = n
  ⟦ e₁ ⟨+⟩ e₂ ⟧e ρ = ⟦ e₁ ⟧e ρ + ⟦ e₂ ⟧e ρ
  ⟦ e₁ ⟨*⟩ e₂ ⟧e ρ = ⟦ e₁ ⟧e ρ * ⟦ e₂ ⟧e ρ

open import Tactic.Nat.Exp  -- <-- full definitions here

module NF (Atom : Set) {{_ : Ord Atom}} where

  open import Data.Bag
  import Tactic.Nat.NF as NF  -- <-- full definitions here

  NF = Bag (List Atom)   -- sum of products: k₁ xy + k₂ xyz + ...

  -- Normalising expressions --
  norm : Exp Atom → NF
  norm (var x) = [ 1 , [ x ] ]
  norm (lit 0) = []
  norm (lit n) = [ n , [] ]
  norm (e ⟨+⟩ e₁) = norm e NF.+nf norm e₁
  norm (e ⟨*⟩ e₁) = norm e NF.*nf norm e₁

  ⟦_⟧t : Nat × List Atom → Env Atom → Nat
  ⟦ k , v ⟧t ρ = k * product (map ρ v)

  ⟦_⟧n : NF → Env Atom → Nat
  ⟦ nf ⟧n ρ = sum (map (flip ⟦_⟧t ρ) nf)

open import Tactic.Nat.NF



--- === Decision procedure proofs === ---

import Tactic.Nat.Auto.Lemmas as Lemmas

module _ {Atom : Set} {{_ : Eq Atom}} {{_ : Ord Atom}} where

  sound : ∀ e (ρ : Env Atom) → ⟦ e ⟧e ρ ≡ ⟦ norm e ⟧n ρ
  sound = Lemmas.sound

  prove : ∀ e₁ e₂ (ρ : Env Atom) → Maybe (⟦ e₁ ⟧e ρ ≡ ⟦ e₂ ⟧e ρ)
  prove e₁ e₂ ρ with norm e₁ == norm e₂
  ... | no  _  = nothing
  ... | yes eq = just $
                   sound e₁ ρ ⟨≡⟩
```

```
                  cong (λ nf → ⟦ nf ⟧n ρ) eq ⟨≡⟩ʳ
                  sound e₂ ρ



--- === Example === ---

Example : Nat → Nat → Set
Example a b = (a + b) ^ 2 ≡ a ^ 2 + 2 * a * b + b ^ 2

mkEnv : List Nat → Env Nat
mkEnv xs n = maybe 0 id (index xs n)

proof₁ : ∀ a b → Example a b
proof₁ a b = fromJust $
  prove ((var 0 ⟨+⟩ var 1) ⟨*⟩ (var 0 ⟨+⟩ var 1))
        (var 0 ⟨*⟩ var 0 ⟨+⟩ lit 2 ⟨*⟩ var 0 ⟨*⟩ var 1 ⟨+⟩ var 1 ⟨*⟩ var 1)
        (mkEnv (a :: b :: []))



--- === Type classes can help === ---

instance
  NumberExp : ∀ {Atom} → Number (Exp Atom)
  NumberExp = record { Constraint = λ _ → ⊤
                     ; fromNat    = λ n → lit n }

  SemiringExp : ∀ {Atom} → Semiring (Exp Atom)
  SemiringExp = record { zro = lit 0
                       ; one = lit 1
                       ; _+_ = _⟨+⟩_
                       ; _*_ = _⟨*⟩_ }

proof₂ : ∀ a b → Example a b
proof₂ a b = fromJust $
  prove ((x + y) ^ 2)
        (x ^ 2 + 2 * x * y + y ^ 2) ρ
  where x = var 0
        y = var 1
        ρ  = mkEnv (a :: b :: [])



--- === Reflection === ---
```

```
open import Builtin.Reflection

-- Primitives --

nameOfNat : Name
nameOfNat = quote Nat

quoteThree : Term
quoteThree = quoteTerm (1 + 2 ofType Nat)

quoteGoalExample : (n : Nat) → n ≥ 0
quoteGoalExample n = quoteGoal g in {!g!}

three : unquote (def nameOfNat [])
three = unquote quoteThree



--- === Using reflection === ---

open import Tactic.Nat.Reflect
open import Tactic.Reflection.Quote

-- fromJust (prove e₁ e₂ ρ)

parseGoal : Term → Maybe ((Exp Var × Exp Var) × List Term)
parseGoal = termToEq

proof-tactic : Term → Term
proof-tactic goal =
  case parseGoal goal of λ
  { nothing → lit (string "todo: error msg")
  ; (just ((e₁ , e₂) , Γ)) →
    def (quote fromJust) $
      vArg (def (quote prove)
        (vArg (` e₁) :: vArg (` e₂) ::
         vArg (quotedEnv Γ) :: [])) :: []
  }

proof₃ : ∀ a b → Example a b
proof₃ a b = quoteGoal g in unquote (proof-tactic g)


--- === Macros === ---

-- macro f : Term → .. → Term
```

```
--   f v₁ .. vn desugars to
--   unquote (f (quoteTerm v₁) .. (quoteTerm vn))



-- proof₃ a b = quoteGoal g in unquote (proof-tactic g)

macro
  magic : Term
  magic = quote-goal $ abs "g" $
          unquote-term (def (quote proof-tactic)
                            (vArg (var 0 []) :: [])) []

proof₄ : ∀ a b → Example a b
proof₄ a b = magic



--- === Wrap up === ---

{-
  Decision procedure: Problem → Maybe Proof
    Only need to compute the 'just' when type checking,
    so you can get good performance.

  Everything is Agda code

  Very thin reflection layer to make it easy to use

  Limitations
    No backtracking (on the meta-level)
    No quasi-quoting
    Untyped reflection

 -}
```