Applications of reflection in Agda

Wouter Swierstra & Paul van der Walt
University of Utrecht

IFL 2012

Intro

- Agda is a functional language with dependent types;
- Since version 2.2.8, Agda has a new *reflection* API;
- Paul is an MSc student at Utrecht;
- His thesis is about having some fun with this new feature.

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
```

Some applications of reflection

- Proof-by-reflection;
- (Well-typed) meta-programming;
- Generic programming;
- Program transformation.

Even

- Suppose we have the following predicate formalizing when a natural number is even:

```
data Even  : ℕ → Set where
  Base : Even 0
  Step : {n : ℕ} → Even n → Even (2 + n)
```

- How can we prove 1024 is even?
  (Example from Adam Chlipala's CPDT)

The dumb approach

The not-so-dumb approach

```
even? : ℕ → Set
even? zero = Unit
even? (suc zero) = Empty
even? (suc (suc n)) = even? n
```

In contrast to the previous predicate, this function *computes*.

Soundness

```
soundness : (n : ℕ) → {p : even? n} → Even n
soundness zero = Base
soundness (suc zero) {()}
soundness (suc (suc n)) {s} =
  Step (soundness n {s})
```

This tells us that to prove `n` is `Even`,
it suffices to show `even? n`;

Soundness

```
soundness : (n : ℕ) → {p : even? n} → Even n
soundness zero = Base
soundness (suc zero) {()}
soundness (suc (suc n)) {s} =
  Step (soundness n {s})
```

This tells us that to prove `n` is `Even`,
it suffices to show `even? n`;
but proving `even? n` is easy for a closed number `n`

Example

```
isEven28 : Even 1024
isEven28 = soundness 1024
```

(Agda fills in the implicit argument of type `Unit` for us)

Non-example

```
isEven28 : Even 13
isEven28 = soundness 13
```

Unresolved implicit argument of type `Empty`

Or if we use an empty type with a more informative name: `IsNotEven 13`

Proof-by-reflection recipe

- Define type of problem domain ($\mathbb{N}$);
- Define predicate of interest (`Even`);
- Define decision procedure (`even?`);
- Prove soundness lemma;
- Profit!

More examples...

- A solver for equations over a some algebraic structure (monoids, rings, ...);
- Automatic procedures for solving problems in some decidable logic (propositional logic, ...)
- (In the paper) a decision procedure for boolean tautologies.

Boolean expressions

Suppose we want to prove:

```
(p1 q1 p2 q2 : Bool) →
  So((p1 ∨ q1) ∧ (p2 ∨ q2)
  ⇒  (q1 ∨ p1) ∧ (q2 ∨ p2))
```

AST

```
data BoolExpr (n : ℕ) → Set where
  Truth     :                                 BoolExpr n
  Falsehood :                                 BoolExpr n
  And       : BoolExpr n → BoolExpr n → BoolExpr n
  Or        : BoolExpr n → BoolExpr n → BoolExpr n
  Not       : BoolExpr n              → BoolExpr n
  Imp       : BoolExpr n → BoolExpr n → BoolExpr n
  Atomic    : Fin n                   → BoolExpr n
```

Evaluation

```
⟦_⊢_⟧ : ∀ {n : ℕ} (e : Env n) → BoolExpr n → Bool
⟦ env ⊢ Truth        ⟧ = true
⟦ env ⊢ Falsehood    ⟧ = false
⟦ env ⊢ And be be₁   ⟧ = ⟦ env ⊢ be ⟧ ∧ ⟦ env ⊢ be₁ ⟧
⟦ env ⊢ Or be be₁    ⟧ = ⟦ env ⊢ be ⟧ ∨ ⟦ env ⊢ be₁ ⟧
⟦ env ⊢ Not be       ⟧ = ¬ ⟦ env ⊢ be ⟧
⟦ env ⊢ Imp be be₁   ⟧ = ⟦ env ⊢ be ⟧ ⇒ ⟦ env ⊢ be₁
⟦ env ⊢ Atomic n     ⟧ = lookup n env
```

Generating all environments

```
forallEnvs : {n : ℕ} → (b : BoolExpr n) → Env n → Set
forallEnvs {0} bexp env =
  if ⟦ env ⊢ Nil ⟧ then Unit else Empty
forallEnvs {suc n} bexp env =
  forallEnvs bexp (true :: env)
  × forallEnvs bexp (false :: env)
```

(Actual implementation is slightly different)

Soundness

- We can now show prove a soundness result that states that:
  - if a boolean expression holds in every environment,
  - then the boolean expression is a tautology.

Calling the solver

- Calling the soundness lemma is a bit more work than we saw previously:

```
foo : (p : Bool) → So (p ⇒ p)
foo = soundness (Imp (Var Fz) ...)
```

- Even if the type already contains all the information we need...

Using reflection

- Agda's `quoteGoal` gives us access to the `Term` representing the type of some unfinished definition;
- We can use this term to generate the required `BoolExpr`.

```
term2boolexpr : (n : ℕ) → (t : Term) → isBoolExprQ' n t → BoolExpr n
term2boolexpr n (con tf []) pf with tf ≟-Name quote true
term2boolexpr n (con tf []) pf | yes p = Truth
term2boolexpr n (con tf []) pf | no ¬p with tf ≟-Name quote false
term2boolexpr n (con tf []) pf | no ¬p  | yes p = Falsehood
term2boolexpr n (con tf []) () | no ¬p₁ | no ¬p
term2boolexpr n (def f (arg a₁ b₁ x ∷ arg a b x₁ ∷ [])) pf | no p
  with f ≟-Name quote _∨_
term2boolexpr n (def f (arg a₁ b₁ x ∷ arg a b x₁ ∷ [])) (proj₁ , proj₂)
| no ¬p | yes p = Or
   (term2boolexpr n x proj₁)
   (term2boolexpr n x₁ proj₂)
....
```

Using the prover

```
not : (b : Bool) → So(b ∨ ¬ b)
not = quoteGoal e in proveTautology e
```

(Note Agda is filling in lots of implicit arguments for us again)

Reflection on reflection

- Some known limitations:
  - no access to function definitions;
  - limited access to data type definitions;
  - no way to generate top-level definitions;
  - untyped!

Can we program with reflection type safely?

Types and contexts

```
data Ty : Set where
  O : Ty
  _=>_ : Ty -> Ty -> Ty

el : U -> Set
el O = Unit
el (s => t) = el s -> el t

Context : Set
Context = List Ty
```

Typed lambda calculus

```
data TypedTerm : Context -> Ty -> Set where
  Lam : TypedTerm (Cons u Γ) v
       -> TypedTerm Γ (u => v)
  App : TypedTerm Γ (u => v) -> TypedTerm Γ u
       -> TypedTerm Γ v
  Var : Ref Γ u -> TypedTerm Γ u
```

Type checking quoted terms

- We can quote an expression to untyped produce a `Term`;
- We can define a function that type checks a `Term` and produces a well-typed term;
- and then manipulate these well-typed terms in some structured way (CPS transform, SKI-translation, ...)

Drawbacks

- We need to write a type checker by hand (not too hard for the simply typed lambda calculus);
- We have to throw away all type information to unquote;
- We still don't get as much type safety as we would like...

Future work

- Generic programming – programming with universes; ornamentation; ...
- Proving termination – well-founded relations, Bove-Capretta method, ...
- Improve the reflection API.