Evidence providing problem solvers in Agda

Uma Zalakain
Conor McBride [S]
Sergey Kitaev [SM]

December 2017

Agda

- Dependently typed
- Functional
- Total
- Proof automation: No tactics, reflection

Theorem proving

- Proposition: type T
  Proof instance: value $v : T$
- Types are checked at compile time:
  Correctness guaranteed statically

Aim

Write programs that compute proofs for:

- Monoids **[implemented]**
- Commutative rings **[understood]**
- Presburger arithmetic **[prelude]**
- Categories **[?]**

Monoids

- A set together with:
  - a binary operation that:
    - is associative
    - has an identity element which is absorbed on either side
- e.g., $(\mathbb{N}, +)$, $(\mathbb{N}, \cdot)$, $\forall \texttt{T}.(\texttt{List T}, \oplus)$
- $x + (y + 0) + ((0 + x) + y) \stackrel{?}{=} ((x + y) + x) + y$
- Lists are their canonical form

Solving a simple monoid, with and without a solver

```
by-solver : (env : Env Nat Nat) → evalExpr nat-monoid env nat-expr-1 ≡ evalExpr nat-monoid env nat-expr-2
by-solver = solve nat-expr-1 nat-expr-2 nat-comp nat-monoid

by-hand : (env : Env Nat Nat) → evalExpr nat-monoid env nat-expr-1 ≡ evalExpr nat-monoid env nat-expr-2
by-hand # =
  ((ε · ε) · ((# 1 · ε) · # 2)) · ((# 1 · # 3) · # 2)
    =[ refl (λ n → (n · ((# 1 · ε) · # 2)) · ((# 1 · # 3) · # 2)) =$= law-ε-· ε >=

  (ε · ((# 1 · ε) · # 2)) · ((# 1 · # 3) · # 2)
    =[ refl (λ n → n · ((# 1 · # 3) · # 2)) =$= law-ε-· ((# 1 · ε) · # 2) >=

  ((# 1 · ε) · # 2) · ((# 1 · # 3) · # 2)
    =[ refl (λ n → ((# 1 · ε) · # 2) · n) =$= law-·-· (# 1) (# 3) (# 2) >=

  ((# 1 · ε) · # 2) · (# 1 · (# 3 · # 2))
    =[ refl (λ n → n · (# 1 · (# 3 · # 2))) =$= law-·-· (# 1) ε (# 2) >=

  (# 1 · (ε · # 2)) · (# 1 · (# 3 · # 2))
    =[ refl (λ n → n) =$= law-·-· (# 1) (ε · # 2) (# 1 · (# 3 · # 2)) >=

  # 1 · ((ε · # 2) · (# 1 · (# 3 · # 2)))
    =[ refl (λ n → # 1 · (n · (# 1 · (# 3 · # 2)))) =$= law-ε-· (# 2) >=

  # 1 · (# 2 · (# 1 · (# 3 · # 2)))
  [QED]
  where open Monoid nat-monoid
```

Equality of canonical forms

Strategy to solve equations on monoids and commutative rings.

- Define:
  - the *source theory* of expressions S
  - an evaluation function $e_S : S \to T$
  - a canonical form N
  - a normalising function $n : S \to N$
  - an evaluation function $e_N : N \to T$
- Proof that $\forall x : S \to e_N\,(n\,x) \equiv e_S\,x$
  Then $\forall xy : S \to n\,x \equiv n\,y \implies e_S\,x \equiv e_S\,y$

Commutative rings

- A set together with:
  - an addition operation that:
    - is associative
    - is commutative
    - has an identity element which is absorbed
    - has an inverse
  - a multiplication operation that:
    - is associative
    - is commutative
    - has an identity element which is absorbed
  - where multiplication is distributive with respect to addition
- e.g., $(\mathbb{N}, +, \cdot)$
- $2 \cdot (2x + 3 \cdot (x + y)) - x \cdot (y + 10) \stackrel{?}{=} (-x + 6) \cdot y$
- Horner normal form together with some normalisation constraints yields a canonical form

Presburger arithmetic

- Theory of natural numbers with addition and equality, logical connectives and existential qualifiers.
- $\forall x.\forall y.\exists z.(x = y + z \wedge (z > 0 \implies x > y))$
- Algorithm:
  - *Fourier-Motzkin* ($\mathbb{R}$, DNF); *Omega Test* ($\mathbb{Z}$, DNF); *Cooper's Algorithm* ($\mathbb{Z}$, no DNF)
  - Proceed eliminating inner quantifiers until none left
  - $\forall x.Px \equiv \neg(\exists x.\neg(Px))$
  - If in $\mathbb{Z}$, then $x \leq y \equiv x < y + 1$

Roadmap

- Decide on which algorithm to use for Presburger. Then:
  1. Translate the algorithm into Agda
  2. Benefit from dependent typing
  3. Prove correctness
  4. Try to make it an addition to Agda-Stdlib
- Better understand the nature and extent of our categorical equation solver. Then:
  1. Decide whether to implement one

Evaluation

- Correctness ultimately guaranteed by the typechecker
- Compare solutions to existing software in other languages
- Apply solvers to real-life problems
- Compare strategies across solvers