Schmitty the Solver

by Wen Kokke

with contributions from Ulf Norell, and hopefully soon, you!

Let's solve some stuff!

```
_ : ∀ x y → x + y ≡ y + x
_ = solveZ3

_ : ∀ x y z → x + (y + z) ≡ (x + y) + z
_ = solveZ3

_ : ∀ x → (x + 2) * (x + -2) ≡ x * x - 4
_ = solveZ3

_ : ∃[ z ] (∀ n → z * n ≡ 0)
_ = solveZ3
```

What if we make a mistake?

```
_ : (x y : ℤ) → x ≤ y → x ≡ y
_ = solveZ3

Found counter-example:
x = + 0
y = + 1
refuting (z : + 0 ≤ + 1) → + 0 ≡ + 1
when checking that the expression unquote solveZ3 has type
(x y : ℤ) → x ≤ y → x ≡ y
```

What is a SMT-LIB?

What is a SMT-LIB?

1. A language for solver input and output.
2. Theories to be supported by solvers: arrays, fixed-size bit vectors, floats, ints, reals, ints and reals, Unicode strings, etc …
3. Logics to be supported by solvers, e.g., "quantifier-free linear integer arithmetic with equality and uninterpreted functions" (or `QF_EUFLIA` for robots)

What is a Schmitty?

What is a Schmitty?

1. an embedding of SMT-LIB in Agda
2. integration with Agda reflection
3. integration with solvers via system calls

What is a Schmitty? — Terms

Sorts, Literals, and Identifiers vary by theory:

```
mutual
  data Term : Set where
    var    : (n : ℕ) → Term
    lit    : (l : Literal) → Term
    app    : (x : Identifier) (xs : Args) → Term
    forAll : (σ : Sort) (x : Term) → Term
    exists : (σ : Sort) (x : Term) → Term

  Args = List Term
```

What is a Schmitty? — Terms

Except… it's well-sorted by construction:

```
mutual
  data Term (Γ : Ctxt) : (σ : Sort) → Set where
    var    : (x : Γ ∋ σ) → Term Γ σ
    lit    : (l : Literal σ) → Term Γ σ
    app    : (x : Identifier Σ) (xs : Args Γ (ArgSorts Σ)) → Term Γ σ
    forAll : (σ : Sort) (x : Term (σ ∷ Γ) BOOL) → Term Γ BOOL
    exists : (σ : Sort) (x : Term (σ ∷ Γ) BOOL) → Term Γ BOOL

  Args : (Γ Δ : Ctxt) → Set
  Args Γ Δ = All (λ σ → Term Γ σ) Δ
```

What is a Schmitty? — Commands & Scripts

There's commands and scripts as well:

```
data Command : Set where
  set-logic     : (l : Logic) → Command
  declare-const : (σ : Sort) → Command
  assert        : Term → Command
  check-sat     : Command
  get-model     : Command

Script = List Command
```

What is a Schmitty? — Commands & Scripts

Except... They're a wee bit more complicated:

```
data Command (Γ : Ctxt) : (δΓ : Ctxt) (δΞ : OutputCtxt) → Set where
  set-logic     : (l : Logic) → Command Γ [] []
  declare-const : (n : String) (σ : Sort) → Command Γ (σ ∷ []) []
  assert        : Term Γ BOOL → Command Γ [] []
  check-sat     : Command Γ [] (SAT ∷ [])
  get-model     : Command Γ [] (MODEL Γ ∷ [])

data Script (Γ : Ctxt) : (Γ′ : Ctxt) (Ξ : OutputCtxt) → Set where
  []   : Script Γ Γ []
  _∷_  : Command Γ δΓ δΞ → Script (δΓ ++ Γ) Γ′ Ξ → Script Γ Γ′ (δΞ ++ Ξ)
```

What is a Schmitty? — Core Theory

```
data CoreSort : Set where
  BOOL : CoreSort

data CoreLiteral : CoreSort → Set where
  -- false and true are identifiers

data CoreId : (Φ : Sig ϕ) → Set where
  false true           : CoreId (Op₀ BOOL)
  not                  : CoreId (Op₁ BOOL)
  implies and or xor : CoreId (Op₂ BOOL)

CoreValue : CoreSort → Set
CoreValue BOOL = Set
-- slightly more complex, due to Type∈Type
```

What is a Schmitty? — Ints Theory

```
data Sort : Set where
  CORE : (ϕ : CoreSort) → Sort
  INT  : Sort

data Literal : Sort → Set where
  core : CoreLiteral ϕ
       → Literal (CORE ϕ)
  nat  : ℕ → Literal INT

Value : Sort → Set
Value (CORE ϕ) = CoreValue ϕ
Value INT      = ℤ
```

```
data Id : (Σ : Sig σ) → Set where

  -- include core identifiers
  core : CoreId Φ → Id (map CORE Φ)

  -- equality, inequality, and ite
  -- are a part of the core theory
  eq neq : Id (Rel INT)
  ite    : Id (BOOL ∷ σ ∷ σ ↦ σ)

  -- theory of integer arithmetic
  not abs             : Id (Op₁ INT)
  sub add mul div mod : Id (Op₂ INT)
  leq lt geq gt       : Id (Rel INT)
```

How does a Schmitty?

(in seven steps)

How does a Schmitty? — ① Reflection

```
_ : ∀ x y → x + y ≡ y + x
_ = solveZ3

  ↓ quoteGoal

_ = pi (vArg (def (quote ℤ) [])) $ abs "x"
  $ pi (vArg (def (quote ℤ) [])) $ abs "y"
  $ def (quote _≡_)
    $ hArg (def (quote Level.zero) [])
    ∷ hArg (def (quote ℤ) [])
    ∷ vArg (def (quote _+_) (vArg (var 1 []) ∷ (vArg (var 0 []) ∷ [])))
    ∷ vArg (def (quote _+_) (vArg (var 0 []) ∷ (vArg (var 1 []) ∷ [])))
    ∷ []
```

How does a Schmitty? – ② Raw Script

```
_ : ∀ x y → x + y ≡ y + x
_ = solveZ3

  ↓ quoteGoal ∘ reflectToRawScript

_ = declare-const "x" (TERM (def (quote ℤ) []))
  ∷ declare-const "y" (TERM (def (quote ℤ) []))
  ∷ assert ( app₁ (quote ¬_) $ app₂ (quote _≡_)
                             $ app₂ (quote _+_) (# 1) (# 0)
                             ∷ app₂ (quote _+_) (# 0) (# 1) )
  ∷ get-model
  ∷ []
```

How does a Schmitty? – ③ Check Script

```
_ : ∀ x y → x + y ≡ y + x
_ = solveZ3

  ↓ quoteGoal ∘ reflectToRawScript ∘ checkRawScript

_ = declare-const "x" INT
  ∷ declare-const "y" INT
  ∷ assert ( app₁ neq
           $ app₂ eq
               $ app₂ add (# 1) (# 0)
               ∷ app₂ add (# 0) (# 1)
           )
  ∷ get-model
  ∷ []
```

How does a Schmitty? – ④ Print Script

```
_ : ∀ x y → x + y ≡ y + x
_ = solveZ3

  ↓ quoteGoal ∘ reflectToRawScript ∘ checkRawScript ∘
    showScript

"(declare-const x_0 Int)
 (declare-const y_1 Int)
 (assert (not (= (+ x_0 y_1) (+ y_1 x_0))))
 (check-sat)
 (get-model)"
```

How does a Schmitty? — ⑤ System Call

```
_ : ∀ x y → x + y ≡ y + x
_ = solveZ3

  ↓ quoteGoal ∘ reflectToRawScript ∘ checkRawScript ∘
    showScript ∘ execTC

"unsat"
```

How does a Schmitty? – ⑥ Parse Outputs

```
_ : ∀ x y → x + y ≡ y + x
_ = solveZ3

  ↓ quoteGoal ∘ reflectToRawScript ∘ checkRawScript ∘
    showScript ∘ execTC ∘ parseOutputs

_ : Sat
_ = unsat -- that’s what we want!
```

How does a Schmitty? — ⑦ Quote Outputs

```
_ : ∀ x y → x + y ≡ y + x
_ = solveZ3

  ↓ quoteGoal ∘ reflectToRawScript ∘ checkRawScript ∘
    showScript ∘ execTC ∘ parseOutputs ∘ quoteOutputs

_ : ∀ x y → x + y ≡ y + x
_ = λ x y → because "z3" (x + y ≡ y + x)
```

What else do we have?

What else do we have?

- Backends for Z3 and CVC4
- Theory of integers linked to Agda integers
- Theory of real numbers linked to Agda floats
- Proofs which compute (when fully applied)

Where to go from here?

Roadmap (easy)

- Add backends for other SMT-LIB compliant solvers
- Add pseudo-sort for naturals to the integer theory
- Add theory of real arithmetic linked to Agda rational numbers
- Add theory of floating-point numbers linked to Agda floats
- Add theory of strings linked to Agda strings
- Add error reporting to the parsers
- Provide witnesses for top-level existentials

Roadmap (moderate)

- Add theory of sequences linked to Agda lists
- Add theory of uninterpreted functions linked to Agda names
- Add theory of regular expressions linked to aGdaREP
- Add theory of algebraic datatypes linked to Agda datatypes
- Add theory of arrays linked to Haskell arrays
- Add support for combined theories
- Add support for logic declarations

Roadmap (hard)

- Add proof checking for Z3 proofs,
  cf. "Proof Reconstruction for Z3 in Isabelle/HOL"

```
unsat
((proof
    (let ((@x36 (monotonicity
        (rewrite (= (= (+ x_0 y_1) (+ y_1 x_0)) true))
        (= (not (= (+ x_0 y_1) (+ y_1 x_0))) (not true)))))
    (let ((@x40 (trans @x36
        (rewrite (= (not true) false))
        (= (not (= (+ x_0 y_1) (+ y_1 x_0))) false))))
    (mp (asserted (not (= (+ x_0 y_1) (+ y_1 x_0)))) @x40 false)))))
```