Proof by Reflection and Automation for Boolean Logic

Initial Bachelor Seminar Talk

Alexander Anisimov

Advisers: Christian Doczkal, Gert Smolka
Supervisor: Gert Smolka

Outline

- **Proof by reflection**
  - reification
  - reflection
  - decision procedure
- Analysis
  - proof terms
  - runtime

Proof by reflection[1]

- translate propositions into terms of an inductive type
  - abstraction from the initial problem
  - so called reification
- run a certified decision procedure
- translate the term back
  - i.e. prove, that the abstraction was correctly chosen
  - so called reflection

[1] as explained by Adam Chlipala

Example:
Boolean tautology solver

- why a tautology solver?
  - many boolean goals when working with Ssreflect
- why reflection?
  - constant overhead in proof terms
  - often faster in practice

Outline

- Proof by reflection
  - **reification**
  - reflection
  - decision procedure
- Analysis
  - proof terms
  - runtime

Reification

- elimination of implication and equivalence (rewrite)

```
(a ==> b) = (~~ a || b)
```

```
(a == b) = (a && b) || (~~ a && ~~ b)
```

- computational representation

```
Inductive term :=
  | Var of nat
  | TT
  | FF
  | And of term & term
  | Or of term & term
  | Not of term.
```

Reification

- atomic expressions
  - collected in a dupfree list
  - mapped to variables by their position
- ```
  Ltac allVars xs e :=
    match e with
      | true => xs
      | ?e1 && ?e2 =>
        let xs := allVars xs e1 in allVars xs e2
      | ...
      | _ => addToList e xs
    end.
  ```
- addToList prevents duplicates
- handles everything, that can't be analized further
  ```

Reification

- ```
  Ltac reify vars b :=
    match b with
      | true => constr:(TT)
      | ?A && ?B =>
        let s := reify vars A in
        let t := reify vars B in
        constr:(And s t)
      | ...
      | _ =>let n := lookup b vars in constr:(Var n)
    end.
  ```
- *lookup* maps a list element to its position by syntactic equality
  - unique mapping for a dupfree list
  ```

Outline

- Proof by reflection
  - reification
  - **reflection**
  - decision procedure
- Analysis
  - proof terms
  - runtime

Reflection

- restore the boolean term
  - structure as in the computational representation
  - variables are mapped back to the atomic terms via their positions

```
Definition denote (phi : nat -> bool) :=
  fix denote s :=
    match s with
      | Var x => phi x
      | TT => true
      | FF => false
      | And s t => denote s && denote t
      | Or s t => denote s || denote t
      | Not s => ~~ denote s
    end.
```

The connection between Reification and Reflection

- Reification
  - bool ~> term
  - in Ltac
  - no proofs
- Reflection
  - term ~> bool
  - in Gallina
  - proof, that the reification was correct

```
vars := allVars nil b
s := reify vars b
phi := nth false vars
-----------------------------
denote phi s = b
```

Outline

- Proof by reflection
  - reification
  - reflection
  - **decision procedure**
- Analysis
  - proof terms
  - runtime

Decision procedure *shandec*

- Shannon Expansion:
  - s is a tautology iff.

    $$tautology\ \ s^x_{true}\ \wedge\ tautology\ \ s^x_{false}$$

    for any variable x in s
  - branching on variables instead of operators
- Approach
  - repeated Shannon expansion
  - simplify as far as possible before every branch

The final tactic

- correctness proof of the decision procedure

```
Lemma shandec_denote s phi:
  shandec s = true -> denote phi s = true.
```

- altogether

```
Ltac shannex :=
  match goal with
  | [ |- ?G = true] =>
    let vars := allVars nil G in
    let s := reify vars G in
    exact (shandec_denote
      s
      (fun n => nth false vars n)
      (eq_refl true) (* shandec s = true *)
    )
  end.
```

Outline

- Proof by reflection
  - reification
  - reflection
  - decision procedure
- **Analysis**
  - **proof terms**
  - runtime

firstorder Proof Term

```
Example E00 a b: ~ ~ ((a \/ ~ a) /\ (b \/ ~ b)).
firstorder. Show Proof. Qed.
```

```
(fun (a b : Prop) (H : ~ ((a \/ ~ a) /\ (b \/ ~ b))) =>
 (fun H0 : a \/ ~ a -> b \/ ~ b -> False =>
  (fun (H1 : a -> b \/ ~ b -> False) (H2 : ~ a -> b \/ ~ b -> False) =>
   (fun H3 : b \/ ~ b -> False =>
    (fun H4 : b \/ ~ b -> False =>
     (fun (H5 : b -> False) (H6 : ~ b -> False) =>
      (fun H7 : False => H7) (H6 H5)) (fun H5 : b => H4 (or_introl H5))
       (fun H5 : ~ b => H4 (or_intror H5))) H3)
     ((fun H3 : a -> False => H2 H3)
        ((fun (_ : False -> b \/ ~ b -> False) (H4 : a) =>
          (fun H5 : b \/ ~ b -> False =>
           (fun (H6 : b -> False) (H7 : ~ b -> False) =>
            (fun H8 : False => H8) (H7 H6)) (fun H6 : b => H5 (or_introl H6))
             (fun H6 : ~ b => H5 (or_intror H6))) (H1 H4))
           (fun H3 : False => H2 (fun _ : a => H3)))))
    (fun H1 : a => H0 (or_introl H1)) (fun H1 : ~ a => H0 (or_intror H1)))
   (fun (H0 : a \/ ~ a) (H1 : b \/ ~ b) => H (conj H0 H1)))
```

shandec Proof Terms

```
Example E01 a b: ((a || ~~ a) && (b || ~~ b)).
shannex. Show Proof. Qed.
```

```
    (fun a b : bool =>
      shandec_denote
        (fun n : nat => nth false [:: b; a] n)
        (And (Or (Var 1) (Not (Var 1)))
          (Or (Var 0) (Not (Var 0))))
        (eqxx (T:=bool_eqType) true))
```

Outline

- Proof by reflection
  - reification
  - reflection
  - decision procedure
- Analysis
  - proof terms
  - **runtime**

Comparison with tauto & firstorder

formula		*shannex*	*tauto*	*firstorder*
$\bigwedge_{i=1}^{n} (a_i \vee \neg a_i)$	n = 40	**0.5**	2.4	1.9
	n = 60	**0.7**	8.4	3.3
	n = 90	**2.1**	36.2	10.7
$\left(\bigwedge_{i=0}^{n} (a_i \to a_{i+1}) \wedge (\neg a_i \to a_{i+1})\right) \to a_{n+1}$	n = 30	3.4	3.6	**1.4**
	n = 45	10.1	13.6	**4.9**
	n = 60	23.7	36.1	**10.2**
$\left(a_0 \wedge \bigwedge_{i=1}^{n} a_{i-1} \to a_i\right) \to a_n$	n = 30	1.2	0.8	**0.7**
	n = 60	6.1	4.8	**4.3**
	n = 90	21.7	18.9	**15.4**
$\left(a_0 \wedge \bigwedge_{i=1}^{n} a_{i-1} \to a_i\right) \to b$	n = 30	1.5	**0.9**	1.6
	n = 60	8.0	**4.7**	9.8
	n = 90	27.9	**19.4**	38.1

Possible improvement

- branch on the variables that occur the most often
  - for each variable, track the number of its occurrences in a sorted dupfree list
  - in most cases this bookkeeping takes more time than one gains
- split conjunctions
  - faster recognition of non-tautologies
- alternative approach
  - exploit the fact that $tautology\ \ s \Leftrightarrow \neg sat(\neg s)$
  - use efficient satsolver techniques

Source

Adam Chlipala

Certified Programming with Dependent Types (2014)

chapter 15

http://adam.chlipala.net/cpdt/