Combining Verification Methods in Software Development: an Overview of a Research Project at Chalmers

Peter Dybjer

Workshop on Automatic and Interactive Verification
Senri, Japan
18 April 2005

CoVer:
Combining Verification Methods in Software Development

Goal: to build a system for verifying Haskell programs using a combination of

- interactive theorem proving
- automatic theorem proving
- random testing

Acknowledgement: The **Programatica** project at Oregon Graduate Institute in Portland.

Combining Three Research Groups at Chalmers

Programming Logic: Martin-Löf Type Theory. Proof assistant *Agda* with window interface *Alfa*. Automatic proof search using the *Agsy* tool.

Formal Methods: SAT-solvers and *automatic first order logic theorem provers*. Applications especially in hardware verification.

Functional Programming: Lazy functional language *Haskell*. Random testing tool *QuickCheck*.

How to do it?

Combining the languages of three different kinds of tools:

Agda: Proof assistant for constructive type theory: dependent types, total functions.

FOL: Automatic theorem provers for classical first order logic.

Haskell: Lazy functional language with Hindley-Milner types, partial recursive functions. A "real" language, but we use *Haskell core*.

And **QuickCheck** has its own "property language" ...

Three subgroups

Haskell - FOL: To translate a Haskell program into a first order theory of combinators. Call external automatic first order prover (Gandalf, Otter, ...) to prove properties of the Haskell program.

Haskell - Agda: To translate a Haskell program into Agda and use Agda to interactively prove properties of it.

Agda - FOL: To translate suitable Agda types to first order formulas. Call external automatic first order prover (Gandalf, Otter, ...) to prove these formulas. Talk by Thierry Coquand about AgdaLight with FOL-plugin and QuickCheck plugin.

Haskell - FOL

A Haskell program generates a *first order theory of combinators*. We have one binary function symbol for application and one constant for each Haskell function.

The translation is done in two steps:

The Glasgow Haskell Compiler translates Haskell program into a core language program ("the ghc external core").

The CoverTranslator translates a core program into a list of equations between combinator terms.

Work in progress on the representation of types in first order logic, and on proof by induction.

Haskell - Agda

Haskell and Agda has an important common subset (modulo polymorphic programs). Moreover, several ideas how to treat Haskell programs outside this subset:

- Represent systematically general recursive Haskell program $f :: A \to B$ by domain predicate $D :: A \to \mathtt{Set}$ and total function $f' :: (x :: A) \to D\ x \to B$

- Use Agda as a logical framework for a first order theory of combinators

- Monadic translation of Haskell programs into Agda. Instantiate to Maybe-monad or to Identity-monad (or potentially other monads which can deal with general recursion)

Haskell - QuickCheck

A simple example of a property definition is

```
prop_RevRev xs = reverse (reverse xs) == xs
  where types = xs::[Int]
```

To check the property, we load this definition in to hugs and then invoke

```
Main> quickCheck prop_RevRev
OK, passed 100 tests.
```

Haskell - QuickCheck (2)

Another QuickCheck property

```
prop_Insert x xs = ordered xs ==> ordered (insert x xs)
  where types = (x::Int, xs::[Int])
```

and another

```
prop_Insert2 x = forAll orderedList $ \xs -> ordered (insert x xs)
  where types = x::Int
```

Haskell - QuickCheck (3)

To QuickCheck conditional formulas

```
p ==> q
  where types = (x1::t1,...,xn::tn)
```

1. randomly generate (x1::t1,...,xn::tn)

2. check whether p is true, if not generate new (x1::t,...,xn::tn)

3. check whether q is true, if not we have a counterexample, otherwise we have a successful test

QuickCheck - FOL

QuickCheck properties correspond to formulas in a first order theory of combinators generated by a Haskell program. For example,

```
forAll orderedList $ \xs -> ordered (insert x xs)
where types = x::Int
```

corresponds to the first order formula

$$\forall x.\mathrm{Int}(x) \Rightarrow \forall xs.\mathrm{OrderedList}(xs) \Rightarrow \mathrm{ordered}@((\mathrm{insert}@x)@xs) = \mathrm{True}$$

QuickCheck - Agda

QuickCheck-style properties can also be defined in Agda, using the dependent type system and the Curry-Howard isomorphism. For example,

```
forAll orderedList $ \xs -> ordered (insert x xs)
where types = x::Int
```

corresponds to the type

```
(x :: Int) -> (xs :: OrderedList) -> ordered (insert x xs) = True
```

in Agda. But note the following ...

QuickCheck - Agda (2)

```
(x :: Int) -> (xsp :: OrderedList) -> ordered (insert x xsp) = True
```

where

```
insert :: Int -> OrderedList -> OrderedList
ordered :: OrderedList -> Bool
```

so always

```
ordered xsp = True
```

Moreover, xsp contains proof information, it's not just a list.

We have replaced testing by proving! But we can combine them (Haysahi)!

QuickCheck - Agda (3)

PhD thesis of Qiao Haiyan 2003 supervised by M. Takeyama and PD.

- QuickCheck-plugin using Alfa's plugin interface. There is now also QuickCheck-plugin for AgdaLight (Ulf Norell 2004).
- Random generators written in Agda/Alfa
- Experiments with combining testing and proving in Agda/Alfa:
  - errors in the program
  - errors in the specification
  - errors in the random generator

  all are roughly equally common!

Three definitions of ordered lists

Recursive definition

```
OrderedList = (xs :: [Int], p :: ordered xs = True)
```

Inductive definition

```
Single :: (x :: Int) -> OrderedListHd x
Cons   :: (x :: Int) -> OrderedListHd y ->  Lte x y ->
          OrderedListHd x

Nil    :: OrderedList
ConsHd :: (x :: Int) -> OrderedListHd x -> OrderedList
```

Inductive-recursive definition ...

Inductive-recursive definition of ordered lists

```
Nil  :: OrderedList
Cons :: (x :: Int) -> (xsp :: OrderedList) -> lb x xsp = True ->
        OrderedList

lb :: Int -> OrderedList -> Bool

lb x Nil             = True
lb x (Cons y xsp q) = x <= y
```

Test data generation and inductive definitions

Inductive definitions are generators: "inductively generated". A naive canonical generator for an inductively defined data type is obtained by selecting a constructor at random, and then continue and randomly generate the arguments.

This works for first order datatypes (algebraic datatypes) where constructors are first order functions.

In constructive type theory we have inductively defined families of types ("inductive families"). The same basic principle of choosing a constructor at random works, but we may need to backtrack. Use relationship between Horn inductive families and logic programs (Hagiya and Sakurai 1984).

Generating theorems

Horn clauses corresponding to to the axioms and inference rules of a system due to Lukasiewicz:

```
thm((P => Q) => ((Q => R) => (P => R))).
thm((~P => P) => P).
thm(P => (~P => Q)).
thm(Q) :- thm(P), thm(P => Q).
```

Running the query `thm(X)` on a Prolog implementation, we can obtain theorems (schemas) as solutions for `X`; for example

```
X = (((_A => _B) => (_C => _B)) => _D) => ((_C => A) => _D)
```

Type theory and logic programs

Type theory	Logic programming
Family of sets $P :: D \to \mathtt{Set}$	Predicate P
an introduction rule	a Horn clause
inductive definition of P	logic program defining P

We call an inductive family arising from a logic program a Horn inductive family. This is a subset of the general class of inductive families considered in type theory.

An inductive family of theorems

Formula is an inductively defined set of formulas.

```
Thm :: Formula -> Set = data
   ax1 (p, q, r :: Formula)
   :: Thm ((p => q) => ((q => r) => (p => r)))
 | ax2 (p       :: Formula)
   :: Thm ((-p => p) => p)
 | ax3 (p, q    :: Formula)
   :: Thm (p => (-p => q))
 | mp  (p, q    :: Formula) (x :: Thm p) (y :: Thm (p => q))
   :: Thm q
```

Another connection between inductive families and logic programs

```
nat(zero).
nat(succ(X)) :-  nat(X).
formula(var(P)) :- nat(P).
formula(~P) :- formula(P).
formula(P => Q) :- formula(P), formula(Q).

thm1((P => Q) => ((Q => R) => (P => R)), ax1(P,Q,R))
                    :- formula(P), formula(Q), formula(R).
thm1((~P => P) => P, ax2(P)) :- formula(P).
thm1(P => (~P => Q), ax3(P,Q)) :- formula(P), formula(Q).
thm1(Q, mp(P,Q,X,Y)) :- thm1(P, X), thm1(P => Q, Y).
```

Concluding remarks

- When a set or a family is (Horn) inductively generated we can also randomly generate or recursively enumerate its elements.

- This is a generic technique. A generator can be written for the whole class of Horn inductive families. (Efficiency is not guaranteed, just like in Prolog.)

- The technique does not only apply to dependent type theory. A variant can be used in predicate logic with inductively defined predicates.