Chasing Bottoms$_\perp$ under Cover[a]

Patrik Jansson @ cs.chalmers.se

February 9, 2004

Refactoring Workshop, Kent University, UK

- The current status of the research programme "Cover — Combining Verification Methods in Software Development."
- Some details from work in progress "Chasing Bottoms — a Case Study in Program Verification in the Presence of Partial and Infinite Values" (with Nils Anders Danielsson)

[a] Funded by the Swedish Foundation for Strategic Research

Patrik Jansson — current activities

"... where type theory meets functional programming ..."

- PolyProof: Generic Functional Programs and Proofs
  - Generic programming meets dependent types
  - Prototyping Generic Programming using Template Haskell
- Cover: Combining Verification Methods in Software Development
  - Translating ghc Core to Agda
  - A case study on verification with partial & inf. values
- Director of Studies for Undergraduate Studies @ cs.chalmers.se
  - Bologna process: Chalmers students will from 2004 take a 3–year BSc + 2–year MSc

A puzzle

Which of these Haskell function definitions are equal?

```
f1    True     x =  x
f2    True =  \x -> x
f3 = \True     x -> x
f4 = \True -> \x -> x
```

Cover — combining verification methods

Grant: 1M euro spread over 3 years

- QuickCheck (specifications as executable test cases in Haskell)
- Alfa/Agda (formal proofs in dependent type theory)
- Model Checking (Propositional and First Order Logic Proving)

"Cover staff": 4.5 full time positions

Prof.:	J. Hughes, T. Coquand, P. Dybjer, M. Sheeran	50%
Ass. Prof.:	M. Benke, K. Claessen, P. Jansson	100%
PostDoc:	Andreas Abel, Grégoire Hamon	200%
PhD stud.:	Q. Haiyan, N.-A. Danielsson, ...	100%

QuickCheck

Specifications as executable test cases in Haskell

Koen Claessen and John Hughes

- a property language (embedded in Haskell)
- combinators for test data generators
- a clever implementation performing the tests

```
import Debug.QuickCheck
prop_associative :: Float -> Float -> Float -> Bool
prop_associative x y z = x+(y+z) == (x+y)+z
```

QuickCheck — an example

```
import Debug.QuickCheck
prop_associative :: Float -> Float -> Float -> Bool
prop_associative x y z = x+(y+z) == (x+y)+z

Main> test prop_associative
Falsifiable, after 2 tests:
-0.3333333
3.0
2.0
```

Alfa/Adga/Cayenne

Constructive logic: propositions as types
proofs as values

- Agda is a dependently typed language (and its proof engine)
- Alfa (by T. Hallgren) is an advanced GUI for Agda
- Cayenne (by L. Augustsson) is almost Agda (and a compiler)

All implemented @ cs.chalmers.se

Cover work on Agda

- portability, scalability, standardisation (Jansson, Benke)
- proof search / automation / tactics (Benke)
- extensions: (Coquand)
  - records (first class modules),
  - implicit calculus (Haskell-connection)
  - datatype generic (Benke, Dybjer, Jansson)

Testing and Proving in Dependent Type Theory

Qiao Haiyan (PhD dec 2003, advisor: Peter Dybjer)

- a "mini-Cover" tool implemented as an Agda-plugin
- QuickCheck-like testing inside Agda
- experiments with combining testing/model checking/proving

Conclusion: Test first, to avoid trying to prove Falsity.

Simpler logics — more automation

Koen Claessen and Mary Sheeran

- Propositional Logic
- First Order Logic

Cover connection:

- Generate first order axioms from the Haskell program
- Can find automatic proofs of "non-recursive" properties
- Induction must be done on the meta-level

Translating Haskell to Agda

Using Agda to prove properties of Haskell programs: we must clarify the connection between the languages. Two paths:

- `hs2alfa`: in the Programatica project (Thomas Hallgren)
  - Using pfe gives full control over the language implementation
- `ghcCore2Agda`: in the Cover project (Patrik Jansson)
  - Using ghc means all programs could be handled

Both are incomplete and work in progress.

Experiments (case studies) could indicate which way to go here.

Chasing Bottoms — verification and partial values

All Haskell types are pointed — for each type a there is a least defined element $\bot_a$ ("bottom at a").

Most Haskell types are actually lifted — there is a distinct "extra" bottom element. Examples:

$$\bot_{(a,b)} \neq (\bot_a, \bot_b)$$
$$\bot_{a \to b} \neq \lambda x \to \bot_b$$

Unfortunate implications: "laws" don't hold

- surjective pairing
- η-expansion

The puzzle solved

Which of these Haskell function definitions are equal?

```
f1, f2, f3, f4 :: Bool -> Int -> Int
f1    True     x =  x
f2    True =  \x -> x
f3 = \True     x -> x    -- same as f1
f4 = \True -> \x -> x    -- same as f2
```

Haskell report has case translation rules from (a), (b), up to (s)!

```
f1 ⊥   = \x -> ⊥
f2 ⊥   = ⊥
```

Test your bottoms!

Problem: many rewrites of Haskell programs are not bottom-preserving — how can we know when we got it right?

- `isBottom :: a -> Bool` is definable using unsafe ghc extensions
- we define QuickCheck test data generators to include $\perp$
- using `isBottom` we define (an approximate) "semantic equality"

Conclusions

- Haskell semantics is tricky
- We need approximate semantics. Perhaps: $\bot = \texttt{\textbackslash x->}\ \bot$

  $\bot = (\bot, \bot)$
- Refactoring may benefit from testing (sanity check)
- Proof may benefit from refactoring (chains of equalities)