Programming Languages Meet Program Verification

Peter Dybjer

Chalmers University, Göteborg, Sweden

Seattle, 21 August, 2006

Theme of workshop

Recent work is exploring alternative, **language-based approaches to program verification**. In these approaches, the programming language provides mechanisms which allow the programmer to express, in some way, her knowledge of why her code meets its specification. This knowledge is **connected more intimately to the code than is usually the case for theorem proving approaches**. One commonly used mechanism is dependent types. Specifications are expressed as types, and the programming language allows proofs of those specifications to be expressed as terms inhabiting those types. Pre- and post-conditions of functions are recorded in their input and return types, and the functions require and produce proofs of those conditions as additional inputs and outputs. One exciting possibility is that languages for programming with proofs may enable developers to target a **"continuum of correctness,"** through varying amounts of effort on specification and verification.

Two main themes?

integrated vs external programming logic Curry-Howard, dependent types, etc

continuum of correctness some properties can be proved automatically or tested

The CoVer Project, Chalmers, 2003 - 2005

CoVer = Combining Verification Methods in Software Development.

- System for verifying Haskell programs by testing and proving (automatic and interactive)
- When Programming Languages met Program Verification at Chalmers
- When Haskell (a programming language) met Agda (a program verification system based on constructive type theory)

The CoVer team: Andreas Abel, Marcin Benke, Ana Bove, Koen Claessen, Catarina Coquand, Thierry Coquand, Nils-Anders Danielsson, Peter Dybjer, Grégoire Hamon, John Hughes, Fredrik Lindblad, Patrik Jansson, Ulf Norell, Mary Sheeran

What is Agda?

A proof assistant for dependent type theory developed at Chalmers since 1997 or so. Successor of the ALF system (1990 -). Especially proof by pointing and clicking.
Agda is an implementation of

- Martin-Löf constructive type theory? (with inductive definitions?)
- Martin-Löf's logical framework? (dependently typed lambda calculus with a universe of sets)
- A functional programming language with dependent types?

The CoVer project - Plan

- History (2000 - 2006)
- Contributions
- Scientific issues
- Principal debates
- Socio-scientific issues
- Lessons for the future

A personal view ...

History of CoVer

2000 The Programatica vision. Mark Jones visits Gothenburg.

2001 The CoVer proposal to SSF (Swedish Strategic Research Foundation).

2002 A pre-study. Combining testing and proving for Agda/Alfa.

2003 First steps. Which approach?

2004 Splitting up: into two subgroups, into three subgroups.

2005 Progress. CoVer-translator, AgdaLight, etc.

2006 Project finished. QuickCheck in industry. Agda in Japan.

The Programatica vision (2000)

Mark Jones (2000). Imagine it is 2010!

> *"Do you remember the days when we didn't prove our programs correct?"*

Programatica

In the Programatica project, we are developing a new kind of program development environment that actively supports and encourages its users in thinking about, **stating, and validating key properties of software as an integral part of the programming process**.

Critically, however, our environment will allow property assertions to be annotated with **“certificates”** that provide evidence of validity. By adopting a generic interface, many different forms of certificate will be supported, offering a **wide range of validation options**—from low-cost instrumentation and automated testing, to machine-assisted proof and formal methods. Individual properties and certificates may pass through several points on this spectrum as development progresses, and as higher levels of assurance are required. To complete the environment, a **suite of "property management” tools** will provide users with facilities to browse or report on the status of properties and associated certificates within a program, and to explore different validation strategies.

The CoVer proposal (2001-2002)

Chalmers has three research groups with relevant knowledge for the Programatica project.

Functional Programming Group Haskell. Random testing with QuickCheck.

Programming Logic Group Martin-Löf type theory. Interactive proof with Agda.

Formal Methods Group SAT-solvers, automatic theorem provers for first order predicate logic (FOL). Applications in hardware.

SSF call for research proposals in IT, summer 2001. We proposed to build

- Programatica-like system based on Agda (adding QuickCheck and automatic methods to Agda)
- ... for Haskell!

It got funded (2002)!

Combining testing and proving in Agda/Alfa (2002-2003)

A QuickCheck property

```
prop x = p x ==> q x
  where types = x :: a
```

Corresponding formula in typed predicate logic

$$\forall x : a.px \Rightarrow qx$$

The corresponding type in Agda: a **testable type** provided there is a **generator** for $x :: a$ such that $px = True$

$$(x : a) \rightarrow (px =_{Bool} True) \rightarrow (qx =_{Bool} True)$$

- testing as an aid to proving (Hayashi)
- reasoning about test data generators
- character of testable types

The project starts. Which approach? (2003)

Agda-centered. Embed Haskell somehow in Agda and prove properties using Agda.

Haskell-centered. Prove properties about Haskell-programs by translation into FOL and use off-the-shelf prover (Vampire, Gandalf).

Dependent Haskell. Design extension of Haskell with dependent types. A **partial type theory**!

Disadvantages and uncertainties with all approaches!

Splitting up (2004)

Automatic group Haskell-centered. (FP, FM).

Interactive group Agda-centered. (Proglog)

Not clear how to unify the results of these two groups.

Splitting up again (2004)

Haskell-FOL How to use automatic FOL-prover to prove Agda-theorems?

Agda-FOL How to use automatic FOL-prover to prove Agda-theorems?

Haskell-Agda How to use Agda for doing interactive proofs about Haskell programs?

Haskell, Agda, and FOL. What is known about their relationships?

Haskell - FOL first order combinatory terms from lambda terms (lambda lifting)

Haskell - Agda encoding general recursive language in primitive recursive language

FOL - Agda Curry-Howard. Agda as a logical framework

Agda - FOL Aczel-interpretation (of type theory into first order theory of combinators; cf abstract realizability, per-model)

Haskell - Haskell Core by ghc compiler

A translation project!

First order theory of combinators (Aczel 1974)

Terms (one binary function symbol + two constants)

$$t ::= x \mid t\,t \mid \mathrm{K} \mid \mathrm{S}$$

Propositions (three unary predicate symbols + equality + logical constants)

$$\begin{aligned} \Phi \quad ::= \quad & \mathcal{N}(t) \mid \mathcal{P}(t) \mid \mathcal{T}(t) \mid t = t \mid \\ & \forall x.\Phi \mid \exists x.\Phi \mid \Phi \to \Phi \mid \Phi \& \Phi \mid \Phi \vee \Phi \mid \top \mid \bot \end{aligned}$$

(Alternative:

$$\cdots \mid t =_{\mathcal{N}} t \mid t =_{\mathcal{P}} t$$

gives per-model)

First order theory of combinators

- $s = t$ means that s and t are convertible:

$$\begin{aligned} \mathrm{K}\,x\,y &= x \\ \mathrm{S}\,x\,y\,z &= x\,z\,(y\,z) \end{aligned}$$

- $\mathcal{N}(t)$ means that t is equal to a Church numeral (λ-terms by bracket abstraction). The rules are

$$\begin{aligned} &\mathcal{N}(0) \\ &\mathcal{N}(x) \implies \mathcal{N}(\mathrm{Succ}\,x) \\ &\Phi[0] \implies (\forall x.\,\Phi[x] \implies \Phi[\mathrm{Succ}\,x]) \implies \forall y.\,\mathcal{N}(y) \implies \Phi[y] \end{aligned}$$

Internal propositions and truths

- $\mathcal{P}(t)$ means that t is a code for a proposition. Such codes (internal propositions) are also obtained by Church-style encodings.
- $\mathcal{T}(t)$ means that t is a code for a true proposition.

The interpretation of Martin-Löf type theory in Aczel's first order theory of combinators

Two examples:

$$f : \mathrm{N} \to \mathrm{N} \quad \text{as} \quad \forall x.\, \mathcal{N}(x) \implies \mathcal{N}(f\,x)$$

$$c : \mathrm{N} \times \mathrm{N} \quad \text{as} \quad \exists x.\, \exists y.\, \mathcal{N}(x)\ \&\ \mathcal{N}(y)\ \&\ c = (x, y)$$

Haskell - FOL

Use automatic FOL prover for proving properties of real Haskell programs

Compiler (ghc) translates Haskell programs to core language programs

CoverTranslator translates core programs to first order theories (lambda lifting, case lifting)

Discussion points

- How to translate types?
- How to prove properties by induction automatically?
- Which axioms are most important for Haskell?

Agda - FOL

How to use a FOL-prover to build proofs in Agda? (Earlier tool Agsy - the Agda synthesizer)

AgdaLight a new experimental implementation of Agda with the following goals

- a light and well-documented system suitable for collaborative experimental work
- connection to **external tools** for automatic proofs and tests. Automatically generated proofs of universally quantified propositional formulae

  $$\forall x.P$$

  can be translated into Agda-derivations of types

  $$(x : A) \longrightarrow P^*$$

- **hidden arguments** as alternative to ML-polymorphism. Agda is a "monomorphic" language, but the type-checker can often infer some of the arguments.

Haskell - Agda

Unlike Agda, Haskell has

- partial functions
- general recursion
- lazy data structures
- (reflexive and nested data structures)
- polymorphism

and

- it's a **real** language, not an idealized one!

How can we use Agda for proving properties about Haskell programs. There are several possibilities, but which one should we choose?

Haskell - Haskell core - Agda

Monadic translation of Haskell core into Agda:

identity monad when Haskell programs are sure to terminate

partiality monad when termination is decidable

general recursion monad ... not tried (cf work of Capretta)

Haskell - Agda: Andreas' fairy tale

Once upon a time, in the year two thousend and two of the Lord, on a fair Spring afternoon, **King Haskell of Glasgow**, regent of great countries and many subordinates (including the Hackers in the Great Green Forests), owner of great treasures, old and powerful scripts and uncountable lines of code, fell deeply in love with **Miss Agda, a young and merry virgin from the small village of Gothenburg in the remote Land of the Welldefined Pleasures**. From one day to the other, he could think of nothing but her graceful appearance, her fair countenance and innocent, bubbling laughter. He knew that he would have no rest until he and she lived under one roof and **shared one cover**. His love was of such fervour that he immediately proposed to her and started to prepare the grand wedding. Although the preparations dragged along and many foreseeable complications had to be overcome (how could such a great aristocrat of noble blood marry a simple woman with no noteworthy dowry), the wedding was finally arranged to be held on a bright October day in the year two thousend and four of the Lord.

The moral of the story:

Translating Haskell into Agda for interacttive proving is not feasible. One needs *one* source language to do all the work in. Either one programs and specifies properties in Haskell and lets the properties be automatically tested or proven by a batch tool (compiler like). Or one programs in Agda where one can equip datastructures and functions with informative invariants. Then one does the interactive proof where one has written and can recognize ones programs. The human being must be the front end, so making him the back end of a translator will fail.

Cover achievements

- Combining testing and proving in Agda
- AgdaLight with plug-ins, FOL, QuickCheck,
- QuickCheck improvements, QuickCheck for Erlang.
- CoverTranslator for automatically proving properties of Haskell programs using FOL-prover
- monadic CoverTranslator for proving properties of Haskell interactively using Agda.

Lack of progress: interactive proof of Haskell programs, case studies combining different features. Darcs.

After CoVer (2006)

QuickCheck Application of QuickCheck for Erlang in industry. (Check with John)

Agda 2 Integrated Verification System jointly being developed by Chalmers and CVS-AIST in Japan. Builds on AgdaLight experience (hidden arguments, connecting external tools, light, well-documented system).

CoverTranslator Automatic proofs of properties of Haskell programs? Case studies? Darcs system.

Scientific issues

- What is the logic of Haskell? What is the semantics of Haskell? What is the logic of an idealized lazy functional programming language? What is the appropriate idealized lazy functional programming language?
- How do you implement the logic of Haskell? Can you deal with the whole general recursive lazy language and still benefit from dependent type theory style type-checking for a terminating subset of the language?
- What is the theory of combining proving and testing? Connections between type theory and testing? What is the logical basis of testing?
- Can constructive type theory be a **practical** programming language?

Principal debates

- Shall we work with a real language (like Haskell) or with an idealized language?
- What should be the fundamental architecture of the system? Shall we build a Haskell-centered system or an Agda-centered system?
- Can we use Agda to prove properties about Haskell? (Scientific issues)?
- Can we manually prove interesting properties about translated code (full Haskell to external core)?
- Is it important to be able to reason about non-termination in Haskell?
- Should we maintain advanced user interface (Alfa)?
- Etc

Socio-scientific issues

CoVer - a software project based on front-line research! needs advanced theory as well as programming wizardry! necesarily a collaborative project - it must combine theorety and practice, logic and programming - many people must collaborate.

- culture clash (how do you get the union rather than the intersection of people's knowledge??)
- academic system (financing of people, hierarchy,
- motivations of researchers
- research funding system (SSF)
- when is a problem solved?
- project management

Lessons for the future - my view

We probably tried to do too many new things at once

- combine proving and testing
- encode general recursion in constructive type theory (in a practical way)
- deal with a full real language with a complex structure.

But

Programatica - CoVer vision A good idea, but start with a clean subset of Haskell!

Specify early Define the language and axioms early. Use standard first order logic.

Implementation Try to use Agda as logical framework. Build support for Haskell verification. (Alternatives, use Isabelle or build dedicated prover.)

Testing and proving Experiment with external tools. There is probably much to learn!