`xmonad` in Coq *(Experience Report)*

Programming a Window Manager with a Proof Assistant

Wouter Swierstra
Universiteit Utrecht
w.s.swierstra@uu.nl

Abstract

This report documents the insights gained from implementing the core functionality of `xmonad`, a popular window manager written in Haskell, in the Coq proof assistant. Rather than focus on verification, this report outlines the technical challenges involved with incorporating Coq code in a Haskell project.

Categories and Subject Descriptors D.2.4 [*Software Program Verification*]: Formal methods; D.3.2 [*Programming Languages*]: Functional programming; F.4.1 [*Mathematical Logic*]: Lambda calculus and related systems

Keywords Coq, dependent types, formal verification, functional programming, Haskell, program extraction, interactive proof assistants, xmonad.

1. Introduction

Starting with Martin-Löf [Martin-Löf 1982], researchers have argued that type theory, with its single language for programs, specifications, and proofs, is the perfect framework in which to write verified software. In practice, there are very few software projects written, specified, and verified in this fashion. The purpose of this experience report is to try and identify some of the reasons for this disparity between theory and practice.

This report documents how hard it is to use today's proof assistants in the verification of a real world application. Specifically, this paper documents my experience using the Coq proof assistant [The Coq development team 2004] to implement and verify parts of the `xmonad` window manager [Stewart and Janssen 2007]. This code uses Coq version 8.3pl2 and `xmonad` version 0.10. The code described in this report is publically available from `https://github.com/wouter-swierstra/xmonad`.

Coq has been used for many large proof developments, such as the proof of the Four Color Theorem [Gonthier 2008] or constructive algebra and analysis [Cruz-Filipe et al. 2004]. The challenge that this paper tackles is not so much about doing proofs in Coq, but rather focuses on the development of verified software. You may know how to do proofs in Coq, but what are the technical problems you may encounter when developing verified applications? What links are missing from the verified programming toolchain? And what are the best practices for engineering verified software?

Haskell'12, September 13, 2012, Copenhagen, Denmark.


```
Section Lists.
  Variable (a : Set).
  Inductive List :=
    Nil : List
  | Cons : a → List → List.
  Fixpoint append (xs ys : List) : List :=
    match xs with
      | Nil ⇒ ys
      | Cons x xs ⇒ Cons x (append xs ys)
    end.
  Infix " ++ " := append
    (right associativity, at level 60).
  Lemma append_assoc (xs ys zs : List) :
    xs ++ (ys ++ zs) = (xs ++ ys) ++ zs.
    Proof.
      induction xs as [| x xs IHxs];
      [| simpl; rewrite IHxs];
      reflexivity.
    Qed.
End Lists.
```

Figure 1. Associativity of append

2. Background

Coq is an interactive proof assistant based on the calculus of inductive constructions [Coquand and Huet 1988]. At its heart is a simple functional programming language, Gallina. Furthermore, Coq provides a separate tactic language to write proofs in an interactive, semi-automatic fashion.

All of these features are illustrated in a small example in Figure 1. The code therein defines an inductive type for lists and a function *append* that appends two lists. After introducing the usual ++ notation for the *append* function, we can state a lemma *append_assoc*, that asserts that the *append* function is associative. Finally, the proof of this lemma is done using some of Coq's tactics: *induction*, *simpl*, *rewrite*, and *reflexivity*. Once Coq has type checked and accepted this code, we can be confident that our *append* function really is associative.

But how can we *call* this function? We can call *append* when defining other functions in *Gallina*, but how can we integrate our verified *append* function with the rest of our codebase? Coq provides an *extraction* mechanism [Letouzey 2003] that generates data types and functions in Haskell, OCaml, or Scheme. For example, extracting Haskell code from Figure 1 yields the code in Figure 2.

```
module Main where
import qualified Prelude
data List a = Nil
  | Cons a (List a)
append :: (List a1) → (List a1) → List a1
append xs ys =
  case xs of
    Nil → ys
    Cons z zs → Cons z (append zs ys)
```

Figure 2. Extracted Haskell code

In this fashion, you can use Coq's extraction mechanism to integrate verified functions into larger software projects. There are a handful of examples of non-trivial developments written in this fashion, the CompCert verified C compiler [Leroy 2006] being one of the most notable examples. Most developments use OCaml as the target language of extraction. This methodology is less popular amongst the Haskell community.

This experience report attempts to identify some of the reasons why this style of verification is not more widespread. It does not aim to document the relative merits of formal verification in Coq and existing technology for the verification of Haskell code; nor does it aim to study the usability of proof assistants to verify modern software. Its single purpose is to document the technical challenges of using code extracted from Coq in a larger Haskell codebase, and how these may be overcome.

3. The `xmonad` window manager

The `xmonad` window manager is a Haskell application that helps organize the windows of the applications running on a machine. It is a *tiling* window manager, that is, all the windows are tiled over the screen, with (in principle) no overlap or gaps. With more than 10,000 downloads from Hackage in 2010 alone it ranks as one of the most popular Haskell applications ever [Stewart 2010]. It is hard to give exact figures, but based on data from Hackage downloads and Ubuntu installs, it is safe to say that `xmonad` has tens of thousands of users.

At the heart of `xmonad` there is a pure model of the current list of windows, including a designated window that is currently in focus. It should come as no surprise that this can be modeled nicely in Haskell using a zipper on lists [Huet 1997], resulting in the following structure called a *Stack* in the `xmonad` sources:

```
data Stack a = Stack {focus :: !a
                     ,up    :: [a]
                     ,down  :: [a]}
```

There are several functions to manipulate these zippers. For instance, the following three functions move the focus one step upwards, move the focus one step downwards, and reverse the stack respectively:

```
focusUp :: Stack a → Stack a
focusUp (Stack t (l:ls) rs) = Stack l ls (t:rs)
focusUp (Stack t [] rs)     = Stack x xs []
  where (x:xs) = reverse (t:rs)

focusDown :: Stack a → Stack a
focusDown = reverseStack . focusUp . reverseStack

reverseStack :: Stack a → Stack a
reverseStack (Stack t ls rs) = Stack t rs ls
```

Note that in contrast to the usual zipper definitions, the *focusUp* function 'wraps around' when the first list in the *Stack* is empty. This is the same behaviour that most applications and operating systems exhibit. To select a new active window, you step through a list of all windows; moving past the last window brings you back to the very first window.

On top of the *Stack* data type, there are several other data types representing workspaces and screens. The 'state' of an `xmonad` session keeps track of the workspaces that are visible (and the screen on which to display them), the workspace that is in focus, and those windows that are not tiled but 'float' on top of the tiled windows.

There are numerous functions similar to *focusUp* for manipulating these data structures. Crucially, all these operations are described as *pure* functions on Haskell data types: there is no I/O or interaction with the X server at this point.

All these pure functions and data types are collected in a single module, `StackSet.hs`, in the `xmonad` source. This module takes up about one quarter of the entire codebase and weighs in at about 550 lines of code, much of which is comments and documentation. This experience report will mostly be concerned with the *StackSet* module, but we will briefly cover the rest of the `xmonad` architecture and development methodology.

On top of this pure model, `xmonad` implements a complete window manager. The central type for the rest of the application is built up using monad transformers as follows:

```
newtype X a =
  X (ReaderT XConf (StateT XState IO) a)
```

The *XConf* type contains configuration data such as keyboard shortcuts and colour schemes; the *XState* type represents the state of the current session, including the *StackSet* types. Finally, all the interaction with the X server happens in the IO monad. The remaining files take care of setting up a connection to the X server and translating user commands to operations on the *StackSet* and ultimately requesting that they are executed by the X server.

The implementers of `xmonad` have tried hard to make the application as stable as possible. It has been used as a testbench for several Haskell source code analysis tools, such as Catch [Mitchell and Runciman 2008] and hlint [Mitchell 2010]. The developers use QuickCheck [Claessen and Hughes 2000] in tandem with HPC [Gill and Runciman 2007] and strive to have tests covering every line of code in the core modules. There is a strong tradition of adopting any new technology that has the potential to uncover new errors.

There are several reasons for choosing `xmonad` as the topic of this case study. The code itself is well-documented and has been extensively reviewed and revised. There is a clear separation between the pure, functional core and the rest of the code. Finally, `xmonad` is a *real world* application with a significant user base. How much effort is it to reimplement the *StackSet* module in Coq?

4. Reimplementation in Coq

The goal of this project is to write a drop-in replacement for the *StackSet* module that is extracted from a Coq source file. Since Gallina is a total language, there are two obvious problems:

- Haskell code does not need to pattern match exhaustively. For example, in the where clause of the *focusUp* function there is no case branch for an empty list. The match will not fail at runtime as reversing a non-empty list always yields a non-empty list, yet there is work to be done to convince Coq of this fact.
- Haskell code may use general recursion. Gallina only permits the definition of structurally recursive functions.

The next two subsections briefly explain how these two issues were resolved in the context of this case study.

Pattern matching

In most cases, functions that used non-exhaustive pattern matches could be rewritten in a way that avoided the case expression altogether. For instance, the *focusUp* function above can be reformulated in Coq as follows:

```
Definition focusUp (s : stack) : stack :=
  match s with
    | Stack t (l :: ls) rs ⇒ Stack l ls (t :: rs)
    | Stack t nil rs ⇒
        Stack (hd t (rev rs)) (tail (rev (t :: rs))) nil
  end.
```

Instead of insisting on matching on *rev* $(t :: rs)$, we use Coq's *hd* function that returns the first element of a list, but requires a default value for the case when the argument list is empty. Although the Coq and Haskell versions are equivalent, the Coq version is a bit less legible.

There were a handful of cases where the pattern match was too intricate to rewrite in this style. In those cases, there are several alternative techniques that can be used [Chlipala 2008; McBride 1999]. One relatively simple solution is to add a local definition for the expression on which case analysis is performed and introduce an additional equality argument to each case branch. In our running example, this yields the following code:

```
Definition focusUp (s : stack) : stack.
refine (match s with
  | Stack t (l :: ls) rs ⇒ Stack l ls (t :: rs)
  | Stack t nil rs ⇒
    let revd := rev (t :: rs) in
    match revd
    return (revd = rev (t :: rs) → stack) with
      | nil ⇒ _
      | x :: xs ⇒ fun prf ⇒ Stack x xs nil
    end _
  end).
```

Here we use Coq's tactic language to produce the desired program. Coq's *refine* tactic fills in an incomplete function definition, generating proof goals for any parts of the definition left open using an underscore. In this example, we end up with two subgoals: using the assumption that $nil = rev\ (t :: rs)$ we rule out the possibility of the nil branch by deriving a contradiction; the second subgoal requires a (trivial) proof that $revd = rev\ (t :: rs)$. Both of these subgoals are easy to discharge using Coq's proof tactics. Upon extraction, these propositional equalities are discarded, leaving a program that is very close to the Haskell original.

There is a danger of using too much automation when writing *programs* in this style. In particular, using tactics such as *auto*, that search the context for any value of the desired type, may lead to *semantically incorrect* programs that are *type correct*. For example, if we had left both case branches of the *focusUp* function open and used tactics to discharge the remaining obligations, these tactics could fill in *Stack t nil nil* in both branches. While the resulting code would type check, the function would not behave as intended.

A third alternative, which was used during the initial prototype, employed Coq's Program framework [Sozeau 2007a,b]. The Program framework automatically generates the stronger types for the separate case branches. Using Program, a developer can mark certain case branches as unreachable and decorate a function's arguments and result with propositions, corresponding to assumptions about arguments and guarantees about the result that a function returns. The developer need only write the *computational* fragment of the definition; the Program framework collects a series of proof obligations that must be fulfilled before the function definition is complete. From the program and these proofs, the Program framework generates a single function definition.

```
focusWindow :: (Eq s, Eq a, Eq i) ⇒
  a → StackSet i l a s sd → StackSet i l a s sd
focusWindow w s
  | Just w == peek s = s
  | otherwise =
    maybe s id $ do
      n ← findTag w s
      return $
        until ((Just w ==).peek) focusUp (view n s)
```

Figure 3. The *focusWindow* function

The function generated by the Program framework, however, tends to be quite complex. Post-hoc verification of these functions is hard. Due to *proof relevance* two definitions with the same computational component, but different associated proofs, are not equal. The functions written in this style carry around complicated proof terms that have been generated automatically, but may not be discarded yet.

The Program automation was very useful during development: it automatically collects the assumptions and proof obligations for every case branch, which makes it easier to find suitable preconditions for totality. The final version of the code, however, avoids the use of Program altogether.

General recursion

Fortunately, most functions in the *StackSet* module make very little use of general recursion, but instead traverse and manipulate inductive data structures in a structurally recursive fashion. Nonetheless, there was one function, *focusWindow* in Figure 3, that needed to be rewritten more drastically. The exact details are not important, but the function's definition is included here to give a taste of the programming style in `xmonad`.

The *focusWindow* function tries to move the focus to an argument window. If the argument window happens to be in focus, which is the branch checked by the first guard, nothing happens. Otherwise the function tries to find the argument window across the different workspaces. If this is successful, the *focusUp* function is called until the desired window is in focus; otherwise, no new window is brought in focus.

The problem with this definition is in the use of the *until* function, which is not structurally recursive. There is a reasonably straightforward alternative: rather than moving up until we reach the desired window, we adapt the *findTag* function to compute the number of steps necessary to bring the desired window into focus. The Coq version of the *focusWindow* function no longer needs to use the *until* function, as it knows precisely how many *focusUp* moves are necessary to bring the required window into focus.

5. Making it all work

It takes less than 500 lines of (uncommented) Coq code to redefine the basics of the Haskell *StackSet* module. Unfortunately, the very first version of the code that is extracted from this module is not very satisfactory.

Custom file headers

There is no way to specify which functions and data types should be exported from a Haskell module that is generated by extraction.

This is less of a problem when extracting to OCaml, as this information is stated in a separate `.mli` file. When extracting to Haskell, however, users may want to hide certain definitions or import certain libraries. As a workaround, I use a shell-script that removes the first fifteen lines of the extracted Haskell code and splices in a custom, hand-written Haskell header.

Using Haskell types

The extraction process generates Haskell functions and data types for *all* the definitions and data types. But what if developers want to use Haskell's standard lists rather than the list types generated by extraction? There are several reasons for this choice. Firstly, some fusion rules and optimizations may be specific to Haskell's lists and list operations. Furthermore, generating new versions of the booleans, lists, pairs, and many other data types from the Haskell Prelude, produces unnecessarily verbose extracted code. Most importantly, clients of the *StackSet* module may want to call functions that take lists as an argument—if these functions must work on extracted data types rather than Haskell data types, any call to the extracted function must first convert between Haskell lists and their extracted counterparts.

The extraction mechanism does provide hooks to customize how functions and data types are extracted. For example, the following two commands change how Coq's boolean type *bool* and the Boolean conjunction are extracted:

Extract Inductive *bool* ⇒
`"Bool"` [`"True"` `"False"`].
Extract Constant *andb* ⇒ `"(&&)"`.

Instead of generating a new data type, Coq's *bool* type is mapped to the Haskell *Bool* type. The first constructor of the *bool* type is mapped to the Haskell constructor *True*; the second constructor of the *bool* type is mapped to the Haskell constructor *False*.

This customization process is extremely fragile. If we swap the mapping of the constructors as follows:

Extract Inductive *bool* ⇒
`"Bool"` [`"False"` `"True"`].

We now map Coq's *true* constructor to *False* and Coq's *false* constructor to *True*. This error will result in extracted code that still type checks, but will exhibit unexpected behaviour. Erroneously replacing *andb* with Haskell's (||) function causes comparable problems. These may seem like innocent mistakes—but incorrect usage of extraction will generate incorrect code, even if the original Coq functions have been verified.

It is important to emphasize that, in principle, the extraction mechanism is guaranteed to preserve a program's semantics [Letouzey 2004]. Incorrect extraction customizations, however, may lead to incorrect programs.

Superfluous coercions

The extraction process inserted several superfluous calls to the *unsafeCoerce* function when it was unsure whether or not the extracted code will typecheck. This turned out to be a bug in the extraction mechanism [Letouzey 2011], that has been fixed in the latest Coq release. The extracted code does not use any unsafe Haskell functions.

Type classes

The original Haskell *StackSet* module defines several functions that use type classes. For example, the *member* function that checks whether or not an argument window is present in a *StackSet* has the following type:

$$member :: Eq\ a \Rightarrow a \to StackSet\ i\ l\ a\ s\ sd \to Bool$$

Although Coq has type classes [Sozeau and Oury 2008], the implementation of instance resolution is completely different from Haskell. Although it is in principle possible to use Coq's type classes, the extraction mechanism is oblivious to their existence. There is no way to generate extracted code that uses type classes. To resolve this, the Coq version of the *StackSet* module starts as follows:

Variable $(a : Set)$.
Variable *eqa* :
$forall\ (x\ y : a), \{x = y\} + \{x <> y\}$.

This declares a type variable a and assumes a decidable equality on a. When extracting a Coq function that uses *eqa*, the generated Haskell function expects an additional argument of type $a \to a \to Bool$ that is used in place of *eqa*. For example, the Coq version of the *member* function yields a Haskell function of type

$$\_member :: (a \to a \to Bool) \to a \to StackSet\ i\ l\ a\ s\ sd \to Bool$$

upon extraction. Functions that do not use *eqa* are not affected by these declarations. To obtain the original type of the Haskell function, we need to define additional wrapper functions that call the extracted functions with suitable 'dictionary' arguments:

$$member :: Eq\ a \Rightarrow a \to StackSet\ i\ l\ a\ s\ sd \to Bool$$
$$member = \_member\ (==)$$

These wrapper functions are all defined in the hand-written header file mentioned previously.

Axioms

The *StackSet* module uses *Data.Map*, Haskell's library for finite maps. For a complete Coq version of the *StackSet* module, we would need to reimplement this library in Coq. However, it is hard to ensure that the extracted code is as efficient as the (heavily optimized) *Data.Map*. Although there are Coq versions of many OCaml data structures, such as finite sets [Filliâtre and Letouzey 2004], writing efficient Haskell code may require pragmas and annotations that are impossible to generate through extraction from Coq alone.

Instead, we add several axioms postulating the existence of finite maps and operations on them:

Axiom $DataMap : Set \to Set \to Set$.
Axiom $empty : forall\ k\ a, DataMap\ k\ a$.
Axiom $insert : forall\ (k\ a : Set),$
$k \to a \to DataMap\ k\ a \to DataMap\ k\ a$.
Axiom $remove : forall\ (k\ a : Set),$
$k \to DataMap\ k\ a \to DataMap\ k\ a$.

Additional extraction commands specify how to generate Haskell code for each of these axioms.

This approach does have its drawbacks: we cannot prove anything about the functions from *Data.Map*. Axioms in Coq do not compute: they have no associated definitions so there is no way to prove anything about their behaviour. The only way to 'prove' properties of axioms, is by adding further axioms stating how various operations on finite maps interact.

After postulating these axioms, there is still more work to be done. The type of the *insert* function from *Data.Map* is actually:

$$insert :: Ord\ k \Rightarrow k \to a \to Map\ k\ a \to Map\ k\ a$$

The type class constraint makes it slightly different from the axiom we have postulated in Coq above. As a result, replacing the axiom with the Haskell *insert* function from *Data.Map* leads to type incorrect code. To fix this, we could, once again, add additional wrapper functions. A more pragmatic solution employed in this

project uses the Unix `sed` tool to insert type class constraints in the type signatures of a handful of functions in the extracted Haskell code.

Patching `xmonad`

After completing all the above steps, the extracted code is almost a drop-in replacement for the original *StackSet* module. A small patch is still needed to the `xmonad` sources to compile using the extracted code. The motivation for this patch requires a bit more explanation about the *StackSet* module.

The *StackSet* module uses a counter to assign unique identifiers to new windows. The type of this counter is kept abstract: it must be an instance of the *Integral* class, even if it is only ever instantiated to *Int*. In principle, we could parametrize our development over the required functions from the *Integral* class, much in the same style as we did for the *eqa* function. As the *Integral* class and its superclasses support quite a few operations, the corresponding wrapper functions would expect equally many arguments. Instead, the Coq version simply instantiates these counters to natural numbers, that in turn are mapped to Haskell's *Int* type through extraction. As a result, the clients of the *StackSet* module need to be adjusted—the *StackSet* type takes one fewer argument. A simple patch to the `xmonad` sources is necessary to take this into account.

6. Discussion

Verification

Having completed this development, it now becomes possible to prove QuickCheck properties in Coq. I have already started to do. Unfortunately, many of the QuickCheck properties are not terribly interesting. For instance, all functions that manipulate a *StackSet* should respect the invariant that every window has a unique identifier. To prove that various operations that permute the order of the windows respect this property is somewhat wearisome, but not conceptually challenging. Proving such properties would make an interesting exercise in Coq, but I suspect it would not drastically improve `xmonad` as QuickCheck is already quite good in this particular domain. The 'bugs' that I have encountered so far tend to be problems in the specification: often a property does not hold for every *StackSet*, but only those generated by the `xmonad` test suite. That does not mean these properties do not hold, but rather that they may require additional assumptions that the QuickCheck properties do not make explicit. Although it would be interesting to complete the verification to compare the relative merits of QuickCheck and Coq, this is beyond the scope of this paper.

Results

The extracted code passes the `xmonad` testsuite and runs as well as the original version. This is a very important sanity check. The transcription to Coq could have introduced errors. Or the extraction commands can introduce bugs of their own. Not all of these mistakes would have been caught by Haskell's type system alone. This goes to show that formal verification in this style can complement, but not replace, existing software verification technology.

Did this project uncover any bugs? Yes! There was a subtle bug in the creation of a new *StackSet*. The *new* function is roughly structured as follows:

```
new l wids m | pre l wids m = ...
             | otherwise = error "StackSet.new"
```

It makes certain assumptions about its arguments, as specified by the precondition *pre*; if these assumptions are not valid, an error is thrown. The problem uncovered by this project was that the precondition *pre* was not strong enough: even if *pre* held, the body of the function could still fail. This was never triggered by users or QuickCheck tests as the only calls to *new* satisfied a stronger, sufficient precondition. Even if not all QuickCheck properties have been proven in Coq, we can now be sure that all the functions from the *StackSet* module are total under certain, precisely specified conditions. In a sense, this development proves that every function from the *StackSet* module will never crash or loop unexpectedly.

Lessons learned

How hard is it to replace Haskell code with Coq in the wild? This experience report shows that it is possible in principle, but painful in practice. There are several predictable issues will need to be addressed: general recursion, incomplete pattern matches, and partial functions. The surprising lesson, for me at least, was the amount of effort it required to smooth out all the niggling issues with interfacing with other Haskell libraries, realizing axioms, custom module headers, type classes, and so forth.

The limitations of program extraction Projects like this one rely on extraction to generate executable code. At the same time, this study shows how tricky it can be to extract usable Haskell code from Coq. This is, in part, because Haskell and Gallina are very different languages. Many of the issues encountered above stem from trying to write Gallina code that uses Haskell-specific language features. Similar projects extracting to OCaml have had much better results [Denney 2001; Filliâtre and Letouzey 2004; Leroy 2006]. *Using extraction successfully requires a close tie between the theorem prover and target programming language.*

If we take the idea of programming with proof assistants seriously, perhaps we should compile Gallina directly to machine code, providing an interface to other languages through a foreign function interface. With the exception of a few PhD theses [Brady 2005; Grégoire 2003], the compilation of dependently typed languages is still very much unexplored. Other dependently typed systems, such as Agda [Norell 2007] and Idris [Brady 2011], are more developed in this respect. They have some form of foreign function interface and support 'compilation' via Haskell or C. At the moment, however, these systems are still very experimental. Agda's extraction to Haskell, for example, introduces calls to *unsafeCoerce* at every node in the abstract syntax tree. Clearly this is undesirable for any high-assurance software development.

Engineering verified software How can we reduce the cost of writing software in this fashion? There are several design choices in `xmonad` that could be made differently to make the shift to Coq easier: reducing the usage of type classes; using total Haskell functions whenever possible; restricting the use of general recursion. If the developers of `xmonad` had been more aware of these issues during the initial design, the transcription to Coq could have been less painful. *Making developers aware of how proof assistants work can facilitate the formal verification of their code.*

The seL4 kernel verification project drew a similar conclusion [Derrin et al. 2006; Klein et al. 2009]. Before starting the formal verification, the systems programmers and proof engineers wrote an executable prototype in Haskell. Once the functionality had been fixed, proof engineers could start the verification and the system developers could write the actual implementation.

This approach to formally verified software would work even better using Coq's extraction technology. Starting with an implementation of the pure `xmonad` core and its specification in terms of QuickCheck properties, proof engineers can port the Haskell code to Coq while the application developers write the

interface to the X server. The pure Haskell module provides a clear interface between the worlds of the proof assistant and the (impure) remainder of the code base.

Future work It is rather depressing to reflect on the amount of effort that is still required for such a project. Coq has a very steep learning curve. There is no tool support to automate the translation from Haskell to Coq. There is no way to formulate Coq lemmas from QuickCheck properties automatically, although such a tool does exist for Isabelle [Haftmann 2010]. Furthermore, there is no tool that uses QuickCheck to test that extracted code behaves the same as its original Haskell counterpart. *There is still much work to be done to develop tools that reduce the cost of writing verified software.*

Acknowledgments

I would like to thank Jeroen Bransen, Jelle Herold, Robbert Krebbers, Pierre Letouzey, José Pedro Magalhães, Thomas van Noort, Don Stewart, the members of the Foundations Group at the Radboud University, and the numerous anonymous reviewers their helpful feedback.

References

E. Brady. *Practical Implementation of a Dependently Typed Functional Programming Language*. PhD thesis, Durham University, 2005.

E. Brady. Idris—systems programming meets full dependent types. In *PLPV'11: Proceedings of the 2011 ACM SIGPLAN Workshop on Programming Languages meets Programming Verification*, 2011.

A. Chlipala. Certified programming with dependent types. Available from `http://adam.chlipala.net/cpdt`, 2008.

K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN International Conference on Functional Programming*, 2000.

T. Coquand and G. Huet. The calculus of constructions. *Inf. Comput.*, 76: 95–120, February 1988.

L. Cruz-Filipe, H. Geuvers, and F. Wiedijk. C-CoRN, the constructive Coq repository at Nijmegen. In *Mathematical Knowledge Management*, 2004.

E. Denney. The synthesis of a Java Card tokenization algorithm. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, 2001.

P. Derrin, K. Elphinstone, G. Klein, D. Cock, and M. M. T. Chakravarty. Running the manual: An approach to high-assurance microkernel development. In *Proceedings of the ACM SIGPLAN Haskell Workshop*, 2006.

J.-C. Filliâtre and P. Letouzey. Functors for Proofs and Programs. In *Proceedings of The European Symposium on Programming*, volume 2986 of *Lecture Notes in Computer Science*, 2004.

A. Gill and C. Runciman. Haskell Program Coverage. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*, 2007.

G. Gonthier. Formal proof: the four-color theorem. *Notices of the AMS*, 55 (11):1382–1393, 2008.

B. Grégoire. *Compilation des termes de preuves: un (nouveau) mariage entre Coq et OCaml.* PhD thesis, Université Paris 7, 2003.

F. Haftmann. From higher-order logic to Haskell: there and back again. In *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 155–158, 2010.

G. Huet. The zipper. *Journal of Functional Programming*, 7(05):549–554, 1997.

G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 207–220. ACM, 2009.

X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 42–54, 2006.

P. Letouzey. A new extraction for Coq. *Types for Proofs and Programs*, 2003.

P. Letouzey. *Programmation fonctionnelle certifiée: l'extraction de programmes dans l'assistant Coq.* PhD thesis, Université Paris-Sud, 2004.

P. Letouzey. Personal communication. 2011.

P. Martin-Löf. Constructive mathematics and computer programming. *Studies in Logic and the Foundations of Mathematics*, 104, 1982.

The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. URL `http://coq.inria.fr`.

C. McBride. *Dependently typed functional programs and their proofs*. PhD thesis, University of Edinburgh, 1999.

N. Mitchell. *HLint Manual*, 2010.

N. Mitchell and C. Runciman. Not all patterns, but enough: an automatic verifier for partial but sufficient pattern matching. In *Proceedings of the first ACM SIGPLAN Symposium on Haskell*, 2008.

U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

M. Sozeau. Program-ing Finger Trees in Coq. In *ICFP'07: Proceedings of the 2007 ACM SIGPLAN International Conference on Functional Programming*, pages 13–24, 2007a.

M. Sozeau. Subset coercions in Coq. In T. Altenkirch and C. McBride, editors, *Types for Proofs and Programs*, volume 4502 of *Lecture Notes in Computer Science*, pages 237–252. Springer, 2007b.

M. Sozeau and N. Oury. First-class type classes. In *Theorem Proving in Higher Order Logics*, 2008.

D. Stewart. Popular haskell packages: Q2 2010 report, June 2010. URL `http://donsbot.wordpress.com/`.

D. Stewart and S. Janssen. xmonad: a tiling window manager. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*, 2007.