Xmonad in Coq:
Programming a window manager in a proof assistant

Wouter Swierstra
FP Dag 6/1/2012
Coq Extraction

- At its heart, Coq has a (simply) typed, total functional programming language *Gallina*.
- *Extraction* lets you turn Gallina programs into Caml, Haskell, or Scheme code.
- Extraction discards proofs, but may introduce ‘unsafe’ coercions.
Extraction in action

• There are a only handful of ‘serious’ verified software developments using Coq and extracted code – CompCert being a notable example.

• Why isn’t it more widely used?
xmonad
xmonad

• A tiling window manager for X:
  • tiles windows over the whole screen;
  • automatically arranges windows;
  • written, configured, and extensible in Haskell;
  • has several tens of thousands of users.
xmonad: design principles

Evil X Server

IO monad

ReaderT

Core

StateT

IO monad
Design principles

• Keep the core pure and functional.

• Separate X server calls from internal data types and functions (Model-view-controller).

• Strive for highest quality code.
Current best practices

• Combining QuickCheck and HPC:
  • Write tests;
  • Find untested code;
  • Repeat.
Can we do better?

• Re-implement core xmonad data types and functions in Coq,

• and ensure that the ‘extracted’ code is a drop-in replacement for the existing Haskell module,

• and formally prove (some of) the QuickCheck properties in Coq.
Blood
Sweat
Shell script
What happens in the functional core?
Data types

data Zipper a = Zipper

  { left :: [a]
  , focus :: !a
  , right :: [a]
  }

Example - I

focusLeft :: Zipper a -> Zipper a
focusLeft (Zipper (l:ls) x rs) =
    Zipper ls l (x : rs)
focusLeft (Zipper [] x rs) =
    let (y : ys) = reverse (x : rs)
    in Zipper ys y []
Example - II

reverse :: Zipper a -> Zipper a
reverse (Zipper ls x rs) =
    Zipper rs x ls

focusRight :: Zipper a -> Zipper a
focusRight =
    reverse . focusLeft . reverse
Did I change the program?
Too general types

- The core data types are as polymorphic as possible: `Zipper a` not `Zipper Window`.
- This is usually, but not always a good thing.
- For example, each window is tagged with a ‘polymorphic’ type that must be in Haskell’s `Integral` class.
- But these are only ever instantiated to `Int`.
Totality

• This project is feasible because most of the functions are structurally recursive.

• But there’s still work to do. Why is this function total?

```plaintext
focusLeft (Zipper [] x rs) =
  let (y : ys) = reverse (x : rs)
  in Zipper [] y ys
```
More totality

• One case which required more work.
• One function finds a window with a given id, and then move left until it is in focus.
• Changed to compute the number of moves necessary and move that many steps.
Extraction problems

• The basic extracted code is a bit rubbish:
  • uses unsafeCoerce (too much);
  • uses Peano numbers, extracted Coq booleans, etc.
  • uses extracted Coq data types for zippers;
  • generates ‘non-idiomatic’ Haskell.
Customizing extraction

- There are various hooks to customize the extracted code:
  - inlining functions;
  - using Haskell data types;
  - realizing axioms.
Danger!

• Using \((a = b) \lor (a \neq b)\) is much more informative than \(\text{Bool}\).

• But we’d like to use ‘real’ Haskell booleans:

  Extract Inductive sumbool \(\Rightarrow\) "Bool" [ "True" "False" ].

• Plenty of opportunity to shoot yourself in the foot!
User defined data types

• Coq generated data types do not have the same names as the Haskell original.

• The extracted file exports ‘too much’.

• Solution:

  • Customize extraction.
  • Write a sed script that splices in a new module header & data types.
Interfacing with Haskell

• I’d like to use Haskell’s data structures for finite maps and dictionaries.

• Re-implementing them in Coq is not an option.

• Add the API as Axioms to Coq...

• ... but also need to postulate properties.

• **Diagnosis: axiom addiction!**
Type classes

• Haskell’s function to check if an element occurs in a list:

\[
\text{elem} :: \text{Eq } a \Rightarrow a \rightarrow [a] \rightarrow \text{Bool}.
\]

• A Coq version might look like:

\[
\text{Variable } a : \text{Set}.
\]
\[
\text{Variable } \text{cmp} : \text{forall } (x y : a),
\]
\[
{x = y} + {x <> y}.
\]
\[
\text{Definition elem : } a \rightarrow \text{list } a \rightarrow \ldots
\]
Extracted code

• Extracting this Coq code generates functions of type:

\[ _\text{elem} :: (a \to a \to \text{Bool}) \to \\
    a \to [a] \to \text{bool}. \]

• Need a manual ‘wrapper function’

\[ \text{elem} :: \text{Eq } a \Rightarrow a \to [a] \to \text{Bool} \]
\[ \text{elem} = _\text{elem} (==) \]
More type class headaches

- We need to assume the existence of Haskell’s finite maps:
  
  Axiom FMap : Set -> Set -> Set.

  Axiom insert : forall (k a : Set),

  k -> a -> FMap k a -> FMap k a.

- In reality, these functions have additional type class constraints...
Another dirty fix

• Need another sed script to patch the types that Coq generates:

  s/insert :: /insert :: Ord a1 => /g

• Not pretty...

• **Lesson:** Gallina is not the same as Haskell.
And now...

• Extraction & post-processing yields a drop-in replacement for the original Haskell module.

• That passes the xmonad test suite.
Verification

- So far, this gives us totality (under certain conditions).
- I’ve proven a few QuickCheck properties in Coq.
- Some properties are trivial; some are more work. But this we know how to do!
Conclusions

- Formal verification can complement, but not replace a good test suite.
- Extraction can introduce bugs!
- If you want to do formal verification, but need `sed` to ‘fix’ your code, something is wrong...
Looking ahead

- There is plenty of work to be done on tighter integration between proof assistants and programming languages.
- You don’t want to write all your code in Coq; but interacting with another programming language all happens through extraction.
- What are the alternatives?