xmonad in Coq

Programming a window manager in a proof assistant

Wouter Swierstra
Haskell Symposium 2012
Coq
Coq

• Coq is ‘just’ a total functional language;
• *Extraction* lets you turn Coq programs into OCaml, Haskell, or Scheme code.
• Extraction discards proofs, but may introduce ‘unsafe’ coercions.
Demo
Extraction to Haskell is not popular.
xmonad

• A tiling window manager for X:
  • arranges windows over the whole screen;
  • written and configured in Haskell;
  • has several tens of thousands of users.
tmp
todo
tmp
trash
wikidoc
xmonad.errors
xmonad.hi
xmonad.hs
xmonad.o
xmonad-x86_64-linux
vav@sibelius ~ $
xmonad: design principles
This paper

- This paper describes a reimplementation of xmonad’s pure core in Coq;
- Extracting Haskell code produces a drop-in replacement for the original module;
- Documents my experience.
Does it work?
Blood
Sweat
Shell script

```
s/delete :: /delete :: Ord a3 => /g
s/remove0 :: /remove0 :: Ord a1 => /g
s/insert :: /insert :: Ord a1 => /g
s/sink :: /sink :: Ord a3 => /g
s/float :: /float :: Ord a3 => /g88d87
< StackSet (Window) Window ScreenId ScreenDetail
  ghc-options: -Werror
  23c23
< en > ScreenId(...), ScreenDetail(...), XState(...),
... = S Int deriving (Eq,Ord,Show,Read,Enum,Num,Integral,Real)
< ScreenDetail(...), XState(...),
109c109
< type WindowSet = StackSet WorkspaceId (Layout Window) Window ScreenId ScreenDetail
  filter (nullScreen)
---
< type WindowSet = StackSet WorkspaceId (Layout Window) Window ScreenId ScreenDetail
115,117d114
< filterStack (notElem v18)
< -- Physical screen indices
< newtype ScreenId :: S Int deriving (Eq,Ord,Show,Read,Enum,Num,Integral,Real)
<
131,132c131,132
< window -> X (Maybe WorkspaceId)
< >>> W.filter (\M.notMember\ W-floating ws)
< (window -> X (ScreenId,>>> W.filter (\notElem\ vis)
```
What happens in the functional core?
Data types

data Zipper a = Zipper

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

... and a lot of functions for zipper manipulation
Totality

- This project is feasible because most of the functions are structurally recursive.
- What about this function?

```haskell
focusLeft (Zipper [] x rs) =
    let (y : ys) = reverse (x : rs)
    in  Zipper ys y []
```
Extraction

• The basic extracted code is terrible!
  • 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;
  - realizing axioms;
  - using Haskell data types.
Interfacing with Haskell

• We would like to use ‘real’ Haskell booleans

\[
\text{Extract Inductive bool } \Rightarrow \\
\text{"Bool" ["True" "False"]}.
\]

• Lots of opportunity to shoot yourself in the foot!
Better extracted code

• The extracted file uses generated data types and exports ‘too much’

• Solution:
  • Customize extraction to use hand-coded data types.
  • Write a `sed` script that splices in a new module header and data type definitions.
Type classes
Type classes

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

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

- A Coq version might look like:

\[
\text{Variable } a : \text{Set}. \\
\text{Variable } \text{cmp} : \text{forall } (x \ y : a), \\
\quad \{x = y\} + \{x <> y\}. \\
\text{Definition } \text{elem} : a \to \text{list } a \to \ldots .
\]
• 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’

\[\text{elem} :: \text{Eq } a \Rightarrow a \to [a] \to \text{Bool}\]
\[\text{elem} = \_\text{elem} (==)\]
Further woes

• This doesn’t scale well to ‘bigger’ type classes (like Ord, Integral, ...);

• Interfacing with existing libraries is an even greater pain;

• Additional `sed` scripts to postprocess the generated Haskell ‘solve’ these issues.
Result!

• This proves the core functions are total*

• Fixed a bug in xmonad.

• More than 25% QuickCheck properties formally verified in Coq.

https://github.com/wouter-swierstra/xmonad/

* under certain conditions.
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...