

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

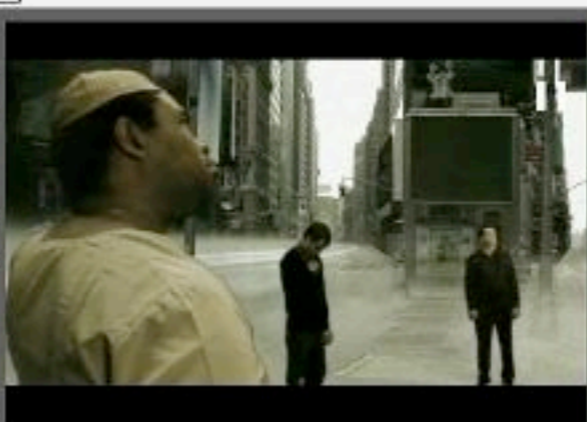
xmonad

- 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.

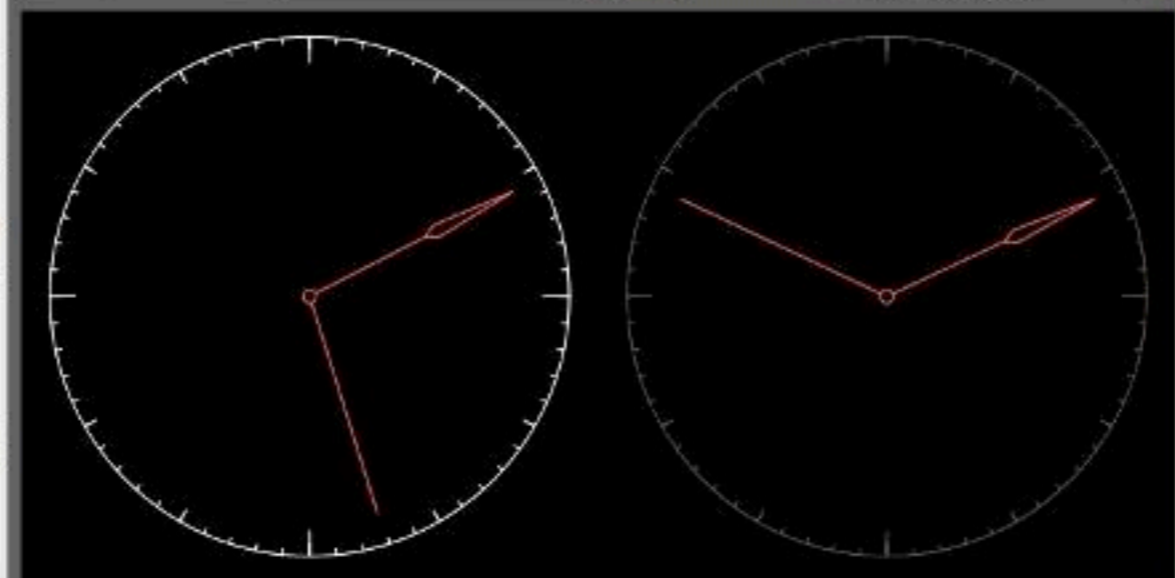
```
File Edit Tools Syntax Buffers Window Help
----
||| named "GridVariants.TallGrid 2 3 (2/3) (4
/3) (5/100)" (GridV.TallGrid 2 3 (2/3) (4/3) (5/100))
||| named "HintedGrid.Grid False" (HGrid.Grid
False)
||| named "HintedGrid.Grid True" (HGrid.Grid
True)
||| named "HintedTile 2 (3/100) (1/2) TopLeft
Tall" (HT.HintedTile 2 (3/100) (1/2) HT.TopLeft HT.Tal
l)
||| named "HintedTile 2 (3/100) (1/2) *Center
* Tall" (HT.HintedTile 2 (3/100) (1/2) HT.Center HT.Tall
)
||| named "HintedTile 2 (3/100) (1/2) *Bottom
Right* Tall" (HT.HintedTile 2 (3/100) (1/2) HT.BottomRig
ht HT.Tall)
||| named "HintedTile 2 (3/100) (1/2) TopLeft
Wide" (HT.HintedTile 2 (3/100) (1/2) HT.TopLeft HT.Wid
e)
||| named "HintedTile 2 (3/100) (1/2) Center
Wide" (HT.HintedTile 2 (3/100) (1/2) HT.Center HT.Wide
)
||| named "HintedTile 2 (3/100) (1/2) BottomR
ight Wide" (HT.HintedTile 2 (3/100) (1/2) HT.BottomRig
ht HT.Wide)
||| named "mastered (3/100) (3/8) $ HintedGri
d.Grid True" (mastered (1/100) (3/8) $ HGrid.Grid True)
||| named "mastered (1/118) (4/9) $ Mirror Tw
oPane"
(mastered (1/118) (4/9) $ Mirror $
TwoPane (3/100) (1/2))
||| named "MosaicAlt" (MosaicAlt M.empty)
)
||| named "Mirror Mosaic [19,5,3]" (Mirror (M
osaic [19,5,3]))
||| named "Mosaic [12,8,5,3,2]" (Mosaic [12,8
,5,3,2])
||| named "Mosaic [4,7]" (Mosaic [4,7]) )
rsLayouts = ResizableTall 2 (1/118) (11/20) [1]
||| named "Spiral (1/2) (default direction, o
rientation)" (spiral (1/2))
||| named "Spiral (golden ratio)" (spiral (to
Rational (2/(1+sqrt(5))::Double)))
||| named "StackTile 1 (3/100) (2/3)" (StackT
ile 1 (3/100) (2/3))
tLayouts = (named "Tall 1 (1/118) (5/9)" (XMonad.Tall
1 (1/118) (5/9))
||| named "ThreeCol 1 (1/118) (1/2)" (ThreeCo
l 1 (1/118) (1/2))
--
||| named "ThreeColumnsMiddle 1 (1/118) (1/
2)" (ThreeColMid 1 (1/118) (1/2))
||| named "TwoPane (3/100) (5/9)" (TwoPane (3
/100) (5/9))
||| named "Mirror TwoPane (3/100) (2/3)" (Mir
ror $ TwoPane (3/100) (2/3)) )
||| simpleTabbed
borHintLayouts = boringWindows $
@
xmonad.hs [haskell][unix] 211/416,85 52%
```

tmp
TODO
trash
wikidoc
xmonad.errors
xmonad.hi
xmonad.hs
xmonad.o
xmonad-x86_64-linux
vav@sibelius ~ \$

SEEKING WHAT YOU'VE DONE
whatsoever
whatsoever gives you a view of what changes you've ma
de in your working copy that haven't
yet been recorded. The changes are displayed in d
arcs patch format. Note that --look-
for-adds implies --summary image.
OTHER COMMANDS
revert Revert is used to undo changes made to the workin
g copy which have not yet been
recorded. You will be prompted for which changes y
ou wish to undo. The last revert can
be undone safely using the unrecord command if the
working copy was not modified in the
meantime.
unrecord
Unrecord does the opposite of record so that it r
emoves the changes from patches: active
changes again which you may record or revert late
r. The working copy itself will not
change.
word-record
Personal page cancelled Line 76



File Edit View Go File Edit View Go
DARCS
Example of Use
Places
vav
Desktop



Most Visited . :⊞: ⌵ ⌶ ⌷ ⌸ ⌹ ⌺ ⌻ ⌼ ⌽ ⌾ ⌿ ⓧ ⓩ ⓪ ⓫ ⓬ ⓭ ⓮ ⓯ ⓰ ⓱ ⓲ ⓳ ⓴ ⓵ ⓶ ⓷ ⓸ ⓹ ⓺ ⓻ ⓼ ⓽ ⓾ ⓿ ⓫ ⓬ ⓭ ⓮ ⓯ ⓰ ⓱ ⓲ ⓳ ⓴ ⓵ ⓶ ⓷ ⓸ ⓹ ⓺ ⓻ ⓼ ⓽ ⓾ ⓿
Xmonad/Scr... gmane.com... Octree e... x
Octree example

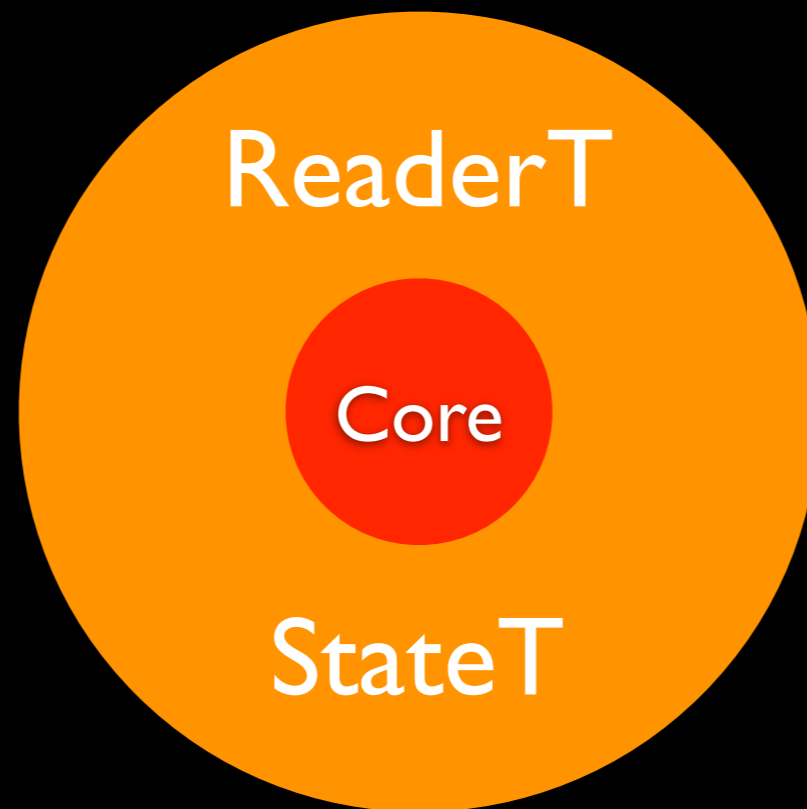
Via #found.
Would you like to comment?
Sign up for a free account, or sign in (if you're already a member).
You Sign in | Create Your Free Account
Explore Places | Last 7 Days | This Month | Popular Tags | The Commons | Creat
Help Community Guidelines | The Help Forum | FAQ | Sitemap | Help by Email
http://www.flickr.com/photos/et [3/3] 90% 0:4

xmonad: design principles

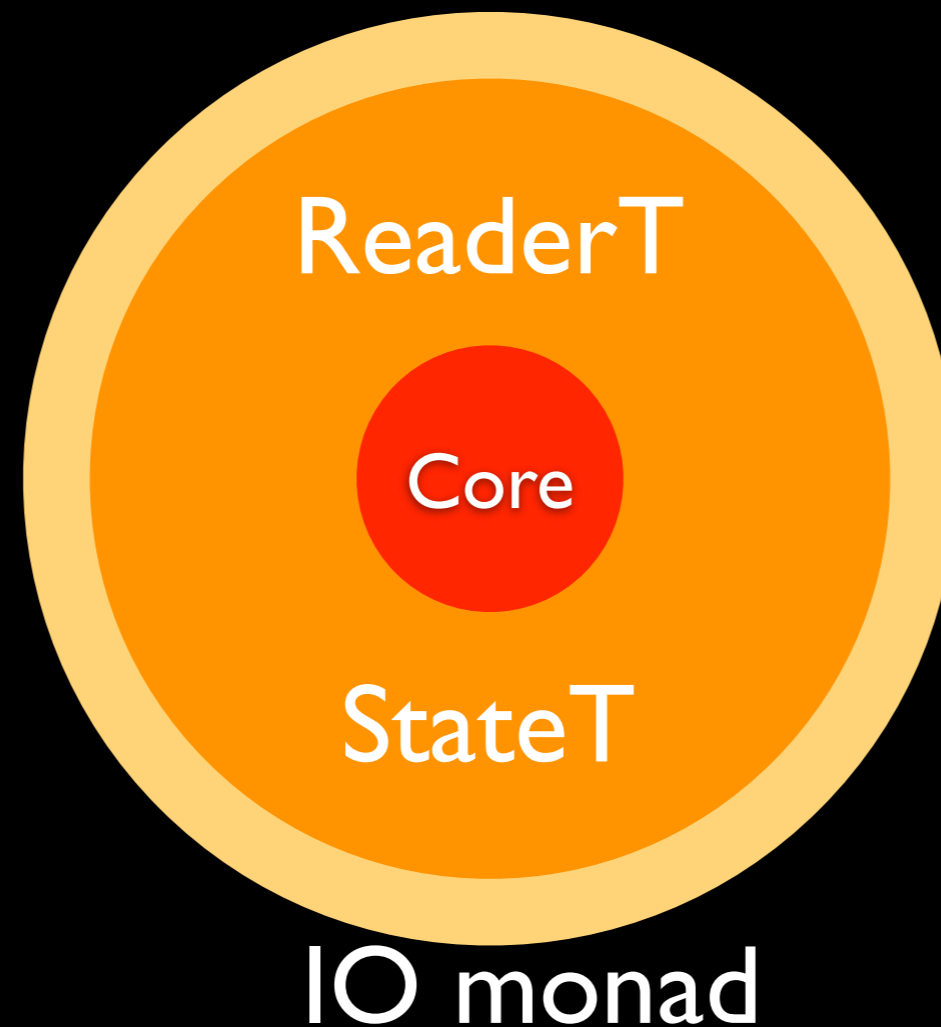
xmonad: design principles



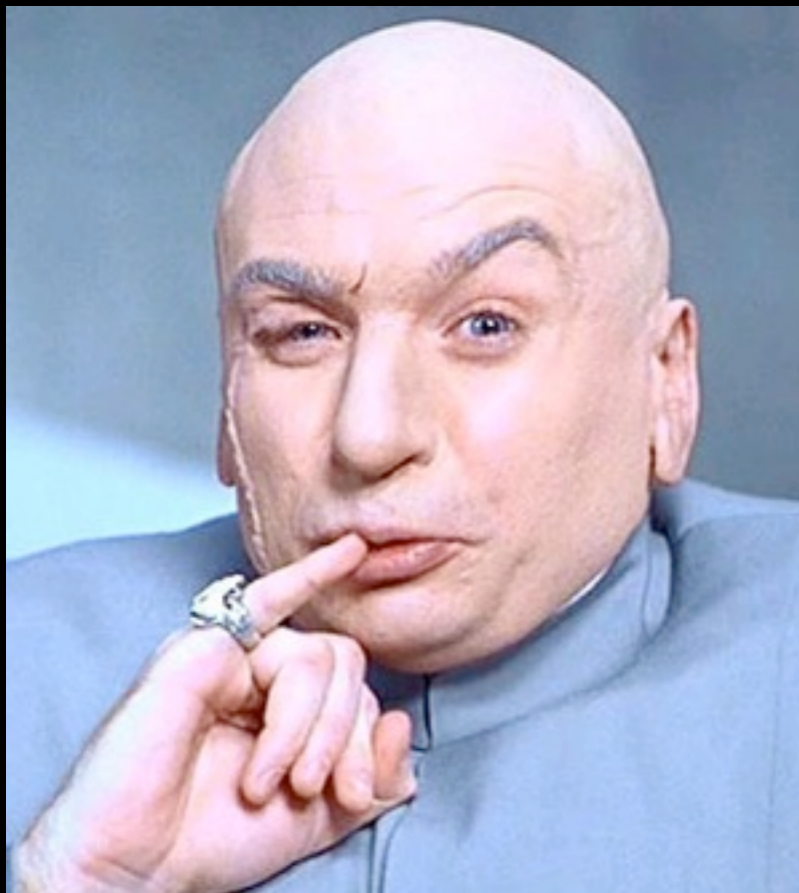
xmonad: design principles



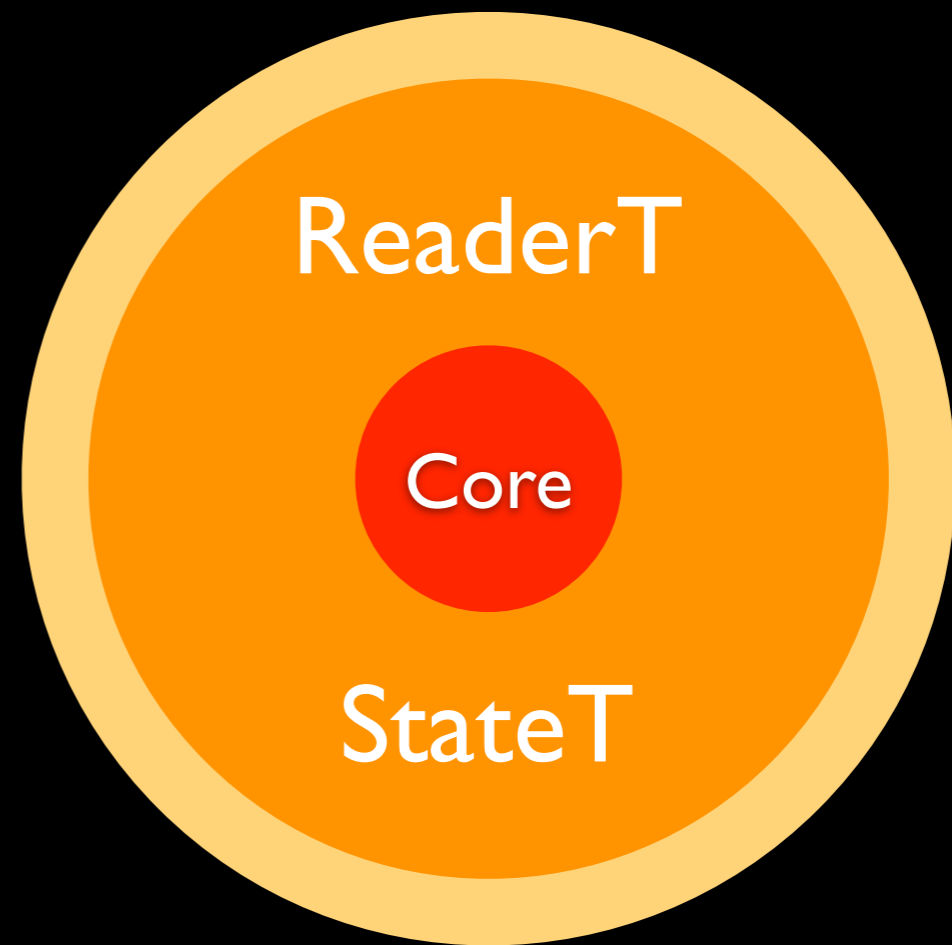
xmonad: design principles



xmonad: design principles



Evil X Server



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

```

1,15d
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 WorkspaceId (Layout Window) Window ScreenDetail
ghc-options: -Werror
23c23
> ScreenId(..), ScreenDetail(..), XState(..),
--- = S Int deriving (Eq,Ord,Show,Read,Enum,Num,Integral,Real)
> ScreenDetail(..), XState(..),
109c109
< type WindowSet = StackSet WorkspaceId (Layout Window) Window ScreenDetail
W.filter ("notElem" vis)
---
> type WindowSet = StackSet WorkspaceId (Layout Window) Window ScreenDetail
115,117d114
< -- | Physical screen indices
< newtype ScreenId = S Int deriving (Eq,Ord,Show,Read,Enum,Num,Integral,Real)
<
131,132c131,132
< X (Maybe WorkspaceId)
>>= W.filter ("notElem" vis)
< Window -> X (ScreenId, W.filter ("notElem" vis))
---

```

Shell script

What happens in the
functional core?

Data types

```
data Zipper a = Zipper
  { left  :: [a]
  , focus :: !a
  , right :: [a]
  }
```

Example - 1

```
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 [] y ys
```

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?

```
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 $(a = b) \vee (a \neq b)$ is much more informative than Bool.
- But we'd like to use 'real' Haskell booleans:

```
Extract Inductive sumbool =>  
"Bool" [ "True" "False" ].
```

Danger!

- Using $(a = b) \vee (a \neq b)$ is much more informative than Bool.
- But we'd like to use 'real' Haskell booleans:

```
Extract Inductive sumbool =>  
"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.

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:

```
elem :: Eq a => a -> [a] -> Bool.
```

- A Coq version might look like:

```
Variable a : Set.
```

```
Variable cmp : forall (x y : a),
```

```
  {x = y} + {x <> y}.
```

```
Definition elem : a -> list a -> ...
```

Extracted code

- Extracting this Coq code generates functions of type:

```
_elem :: (a -> a -> Bool) ->  
        a -> [a] -> bool.
```

- Need a manual ‘wrapper function’

```
elem :: Eq a => a -> [a] -> Bool  
elem = _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?