# Adventures in Extraction

Wouter Swierstra

Brouwer Seminar, 28/3/2011

with some slides from Don Stewart

ex•trac•tion |ikˈstrak SH ən|

noun

**1** the action of taking out something, esp. using effort or force :

*mineral extraction | a dental extraction.*

# Coq Extraction

- At its heart, Coq has a (simply) typed mini-programming language *Gallina*.
- *Extraction* lets you turn Gallina programs into Caml, Haskell, or Scheme code.

Inside every proof assistant, there's a functional language struggling to get out.

**Idea:** Extraction lets you write verified software in a heterogeneous programming environment.

# 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?

# This talk

- An *experience report* documenting an attempt at using extraction to replace a non-trivial Haskell program.
- An attempt to identify the software engineering principles of verification.

# xmonad

# xmonad

- A tiling window manager for X:
  - tiles windows over the whole screen;
  - automatic arranges windows;
  - written, configured, and extensible in Haskell;
  - had more than 10k downloads in 2010.


DELL
DELL
DELL
Graham Hutton
Programming in Haskell

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.Tall)
           ||| 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) *BottomRight* Tall" (HT.HintedTile 2 (3/100) (1/2) HT.BottomRight HT.Tall)
           ||| named "HintedTile 2 (3/100) (1/2) TopLeft Wide"  (HT.HintedTile 2 (3/100) (1/2) HT.TopLeft HT.Wide)
           ||| 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) BottomRight Wide"   (HT.HintedTile 2 (3/100) (1/2) HT.BottomRight HT.Wide)
           ||| named "mastered (3/100) (3/8) $ HintedGrid.Grid True" (mastered (1/100) (3/8) $ HGrid.Grid True)
           ||| named "mastered (1/118) (4/9) $ Mirror TwoPane"
                     (mastered (1/118) (4/9) $ Mirror $ TwoPane (3/100) (1/2))
           ||| named "MosaicAlt" (MosaicAlt M.empty)
           )
----       ||| named "Mirror Mosaic [19,5,3]" (Mirror (Mosaic [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, orientation)" (spiral (1/2))
           ||| named "Spiral (golden ratio)" (spiral (toRational (2/(1+sqrt(5)::Double))))
           ||| named "StackTile 1 (3/100) (2/3)" (StackTile 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)" (ThreeCol 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)" (Mirror $ TwoPane (3/100) (2/3)) )
           ||| simpleTabbed

borHintLayouts = boringWindows $
a
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 ~ $
```

```
SEEING WHAT YOU'VE DONE
       whatsnew
              whatsnew gives you a view of what changes you've made in your working copy that haven't
              yet been recorded.  The changes are displayed in darcs patch format. Note that --look-
              for-adds implies --summary usage.

OTHER COMMANDS
       revert Revert is used to undo changes made to the working copy which have not yet been
              recorded.  You will be prompted for which changes you wish to undo.  The last revert can
              be undone safely using the unrevert command if the working copy was not modified in the
              meantime.

       unrevert
              Unrevert is used to undo the results of a revert command. It is only guaranteed to work
              properly if you haven't made any changes since the revert was performed.

       unrecord
              Unrecord does the opposite of record in that it removes the changes from patches active
              changes again which you may record or revert later.  The working copy itself will not
              change.

       amend-record
Manual page darcs(1) line 76
```

File Edit View Go File Edit View Go

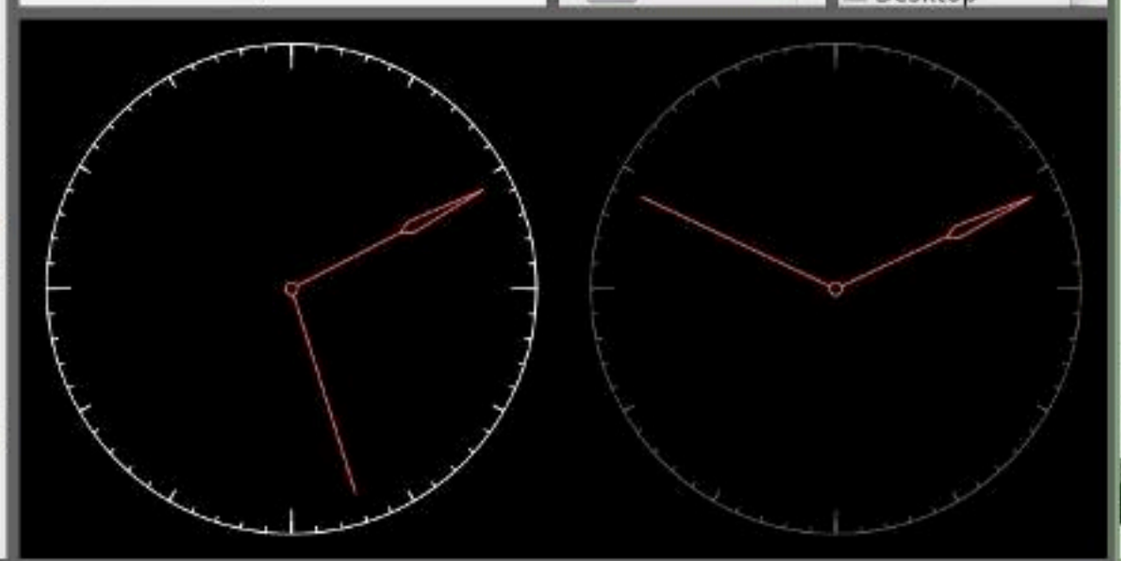
DARCS

Example of Use

Places

vav

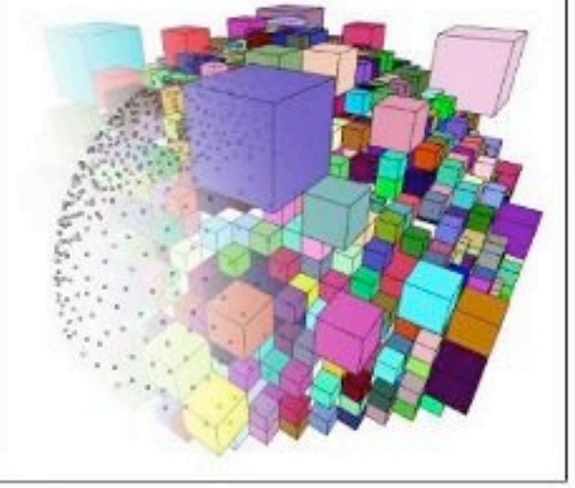
Desktop



Most Visited . :⊗: ɑ ĠŃ2 λ )H( ʘ ¤ X seq »

Xmonad/Scr... | gmane.com... | Octree e... ×

**Octree example**



Via ffffound.

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 | Creati

Help Community Guidelines | The Help Forum | FAQ | Sitemap | Help by Email

http://www.flickr.com/photos/et [3/3] 90% 0:4

[Spiral (golden ratio)]

# Testimonials

Xmonad fits right into how I think window managers should be.

# Testimonials

Xmonad is easily the fastest and has the smallest memory footprint I have found yet.

# Testimonials

Xmonad is by far the best window manager around. It's one of the reasons I stick with Linux.

# Comparison

| tool | loc | Language |
|---|---|---|
| metacity | > 50k | C |
| ion | 27k | C |
| ratpoison | 13k | C |
| wmii | 7k | C |
| dwm | 1.7k | C |
| xmonad | 2.5k | Haskell |

# xmonad

"That was easy. xmonad rocks!"

Google™ Custom Search | Search

home | download | documentation | community

## What is xmonad?

xmonad is a dynamically tiling X11 window manager that is written and configured in Haskell. In a normal WM, you spend half your time aligning and searching for windows. **xmonad makes work easier**, by automating this.

## What's new?

- xmonad 0.9 is available from our download page.
- Report a bug and we'll squash it for you in the next release.
- Follow our blog or on twitter, or the xmonad reddit.

## Why should I use xmonad?

xmonad is *tiling*.
xmonad automates the common task of arranging windows, so you can concentrate on getting stuff done.
xmonad is *minimal*.
out of the box, no window decorations, no status bar, no icon dock. just clean lines and efficiency.
xmonad is *stable*.
haskell + smart programming practices guarantee a **crash-free** experience.
xmonad is *extensible*.
it sports a vibrant extension library, including support for window decorations, status bars, and icon docks.
xmonad is *featureful*.
core features like **per-screen workspaces**, true xinerama support and managehooks can't be found in any other wm.
xmonad is *easy*.
we work hard to make common configuration tasks one-liners.
xmonad is *friendly*.
an active, friendly mailing list and irc channel are **waiting to help** you get up and running.

## screenshots

see more...

## videos

screencast @ youtube | screencast @ youtube | screencast @ youtube

view more...

## Testimonials

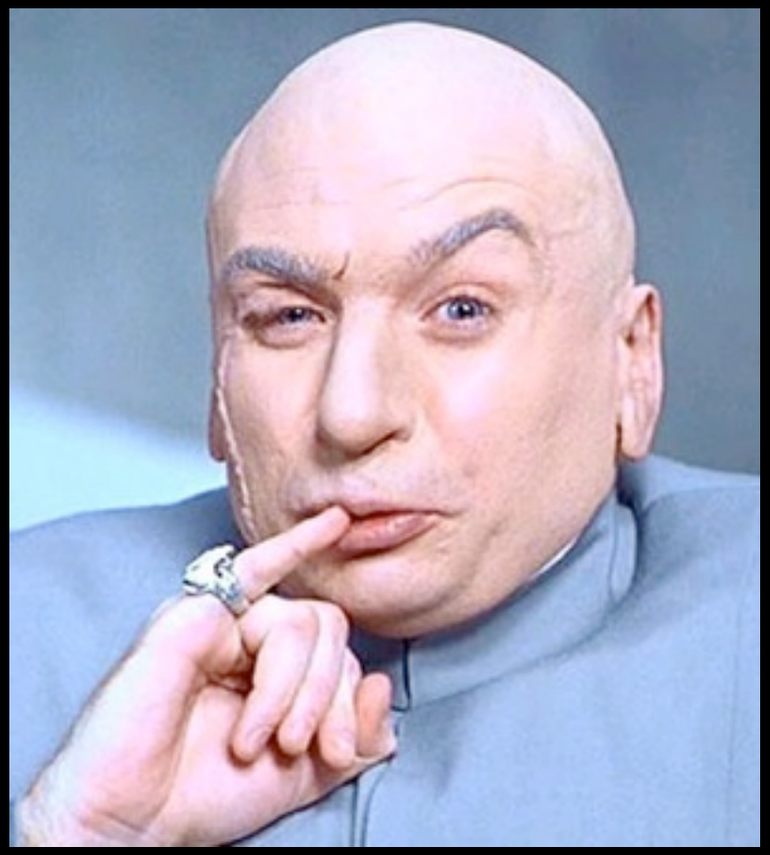
"Multimonitor xmonad has to be the best Linux desktop experience ever" — josephkern apr 09

"XMonad is by far the best Window Manager around. It's one of the reasons I stick with Linux." — Tener, apr 09

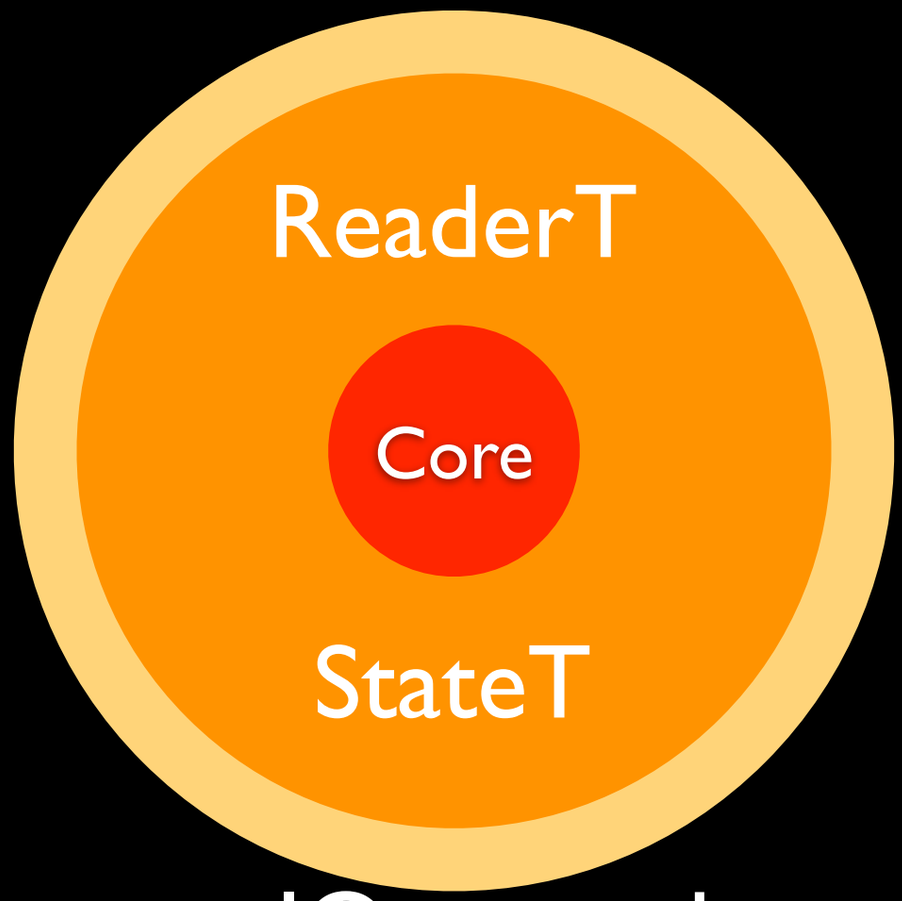
"I have to say, the greatest thing about xmonad thus far is its insane stability… I have zero issues with xmonad" — wfarr, mar 08

read more...

# 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.

# 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 [] 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
```

# Simplification

- The “real” data types talk about several workspaces, some of which may be hidden, each with their own unique id.
- But these Zipper types are really at the heart of xmonad.

# How can we make sure the code is reliable?

# Reliability toolkit

- Cabal build system;
- Type system;
- -Wall compiler flags;
- QuickCheck;
- HPC.

# QuickCheck

- Given properties that you expect your function to satisfy, QuickCheck generates random input and tries to find a counter example. For instance:

```
zipLeftRight :: Zipper Int -> Zipper Int

zipLeftRight z =

  focusRight (focusLeft z) == z
```

# HPC

- The Haskell Program Coverage tool keeps track of which expressions are evaluated during execution.
  - dead code;
  - spurious conditionals;
  - untested code;
  - ...

# Example report

```
  67% expressions used (72/106)
 14% boolean coverage (1/7)
     16% guards (1/6), 2 always True, 2 always False, 1
unevaluated
      0% 'if' conditions (0/1), 1 always True
    100% qualifiers (0/0)
 42% alternatives used (3/7)
 88% local declarations used (8/9)
 80% top-level declarations used (4/5)
unused declarations:
    position
    showRecip.p
```

# HTML report

```haskell
reciprocal :: Int -> (String, Int)
reciprocal n | n > 1 = ('0' : '.' : digits, recur)
  where
  (digits, recur) = divide n 1 []

divide :: Int -> Int -> [Int] -> (String, Int)
divide n c cs | c `elem` cs = ([], position c cs)
              | r == 0      = (show q, 0)
              | r /= 0      = (show q ++ digits, recur)
  where
  (q, r) = (c*10) `quotRem` n
  (digits, recur) = divide n r (c:cs)

position :: Int -> [Int] -> Int
position n (x:xs) | n==x      = 1
                  | otherwise = 1 + position n xs

showRecip :: Int -> String
showRecip n =
  "1/" ++ show n ++ " = " ++
  if r==0 then d else take p d ++ "(" ++ drop p d ++ ")"
  where
  p = length d - r
  (d, r) = reciprocal n

main = do
  number <- readLn
  putStrLn (showRecip number)
  main
```

# High-assurance software

- Combining QuickCheck and HPC:
  - Write tests;
  - Find untested code;
  - Repeat.

# Putting it in practice

- xmonad has:
  - ±100% test coverage core functions and data structures;
  - More than 100 automatically checked QuickCheck properties;
  - No new patches accepted until all tests pass and all code is tested.

# But can we do better still...

# What I've done

- Re-implemented core xmonad data types and functions in Coq,
- Such 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
<         ghc-options:     -Werror
23c23
<     ScreenId(..), ScreenDetail(..), XState(..),
---
>     ScreenDetail(..), XState(..),
109c109
< type WindowSet   = StackSet  WorkspaceId (Layout Window) Window ScreenId Scree
nDetail
---
> 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
<                     >>= W.filter (`M.notMember` W.floating ws)
<                     >>= W.filter (`notElem` vis)
---
```

# Shell script

# What I've learned

- Extraction is not yet mature technology.
- Formal verification can complement, but not replace a good test suite.
- There is plenty of work to be done on tighter integration between proof assistants and programming languages.

# 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.

# 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!**

# 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) ∨ (a ≠ 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.

# 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...
- Coq is not the same as Haskell/OCaml.

# 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).
- Several QuickCheck properties have been proven to hold in Coq.
- Some properties are trivial; some are more work. But this we know how to do!

# Conclusions

- Extraction is not yet mature technology.
- If you want to do formal verification, sed should not be a mandatory part of your toolchain.

# Conclusions

- Formal verification can complement, but not replace a good test suite.
- Extraction can introduce bugs!
- Never trust ‘formally verified code’ that hasn’t been tested.

# Conclusions

- 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?