extraction | ikˈstrækʃən |
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

• 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.
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

<table>
<thead>
<tr>
<th>tool</th>
<th>loc</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>metacity</td>
<td>&gt; 50k</td>
<td>C</td>
</tr>
<tr>
<td>ion</td>
<td>27k</td>
<td>C</td>
</tr>
<tr>
<td>ratpoison</td>
<td>13k</td>
<td>C</td>
</tr>
<tr>
<td>wmii</td>
<td>7k</td>
<td>C</td>
</tr>
<tr>
<td>dwm</td>
<td>1.7k</td>
<td>C</td>
</tr>
<tr>
<td>xmonad</td>
<td>2.5k</td>
<td>Haskell</td>
</tr>
</tbody>
</table>
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.
xmonad: design principles

Evil X Server

IO monad

StateT

Core

ReaderT
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

\begin{verbatim}
data Zipper a = Zipper
  { left :: [a]
     , focus :: !a
     , right :: [a]
  }
\end{verbatim}
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:

```haskell
zipLeftRight :: Zipper Int -> Zipper Int
zipLeftRight z =
  focusRight (focusLeft z) == z
```
The Haskell Program Coverage tool keeps track of which expressions are evaluated during execution.

- dead code;
- spurious conditionals;
- untested code;
- ...

HPC
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

```
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 | e elem cs = ([], position c cs)
  where
    (q, r) = (c*10) `quotRem` n
    (digits, recur) = divide n r (c:cs)
    position n (x:xs) | n==x = 1
      otherwise = 1 + position n xs

showRecip :: Int -> String
showRecip n = "1/" ++ show n ++ " = " ++
  where
    (d, r) = reciprocal n

main = do
  number <- readLn
  putStrLn (showRecip number)
```
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
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`. 
This project is feasible because most of the functions are structurally recursive.

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

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

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

\[
\text{Extract Inductive sumbool} \Rightarrow \"\text{Bool}\" \left[ \"\text{True}\" \ \"\text{False}\" \right].
\]

• 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 -> \text{Bool}) ->
  a -> [a] -> \text{bool}.

- Need a manual ‘wrapper function’

  \text{elem} :: \text{Eq} \ a => a -> [a] -> \text{Bool}
  elem = _elem (==)
More type class headaches

• We need to assume the existence of Haskell’s finite maps:

Axiom \text{FMap} : \text{Set} \rightarrow \text{Set} \rightarrow \text{Set}.

Axiom insert : \forall (k \ a : \text{Set}),

\begin{align*}
k \rightarrow a \rightarrow \text{FMap} \ k \ a \rightarrow \text{FMap} \ k \ a.
\end{align*}

• 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?