Automated refactoring

Course „Software Language Engineering“

University of Koblenz-Landau
Department of Computer Science
Ralf Lämmel
Software Languages Team
Motivation
What’s refactoring?

Grammar, 1st version
stm → “if” exp “then” stm “else” stm “fi”
stm → “if” exp “then” stm “fi”

2nd version
stm → “if” exp “then” stm (“else” stm)? “fi”

3rd version
stm → “if” exp “then” stm optional-else “fi”
optional-else → “else” stm
optional-else → ε
What’s manual refactoring?

Grammar, 1st version
stm → “if” exp “then” stm “else” stm “fi”
stm → “if” exp “then” stm “fi”

2nd version
stm → “if” exp “then” stm (“else” stm)? “fi”

3rd version
stm → “if” exp “then” stm optional-else “fi”
optional-else → “else” stm
optional-else → ε
What’s **automated** refactoring?

Grammar, 1st version

\[
stm \rightarrow \text{“if” exp “then” stm “else” stm “fi”}
stm \rightarrow \text{“if” exp “then” stm “fi”}
\]

2nd version

\[
stm \rightarrow \text{“if” exp “then” stm (“else” stm)? “fi”}
\]

3rd version

\[
stm \rightarrow \text{“if” exp “then” stm optional–else “fi”}
\text{optional–else} \rightarrow \text{“else” stm}
\text{optional–else} \rightarrow \varepsilon
\]

The refactoring transformations are automated by tools.

The next step would be to automate **search** for refactoring opportunities.
Program refactoring using functional aspects
Tool Support for Complex Refactoring to Design Patterns
Automated Extract Component Refactoring
Refactoring with Aspects
Reconciling manual and automatic refactoring
Use, disuse, and misuse of automated refactorings
Using structural and semantic information to support software refactoring
Search-Based Software Maintenance
Tool-Supported Refactoring of Existing Object-Oriented Code into Aspects
Automatic Refactoring of Erlang Programs
Automated Inference of Pointcuts in Aspect-Oriented Refactoring
Refactoring for Parameterizing Java Classes
Aspect-Oriented Refactoring of Legacy Applications: An Evaluation
On the use of genetic programming for automated refactoring and the introduction of design patterns
Extreme Product Line Engineering - Refactoring for Variability: A Test-Driven Approach
Refactoring Access Control Policies for Performance Improvement
Towards Generic Refactoring :-)
A very simple refactoring: Inlining lets in a simple functional language

https://github.com/slecoursel/slecoursel/tree/master/sources/refactoring/
Concrete syntax of FL

\[
mult \ n \ m = \text{if (} n == 0 \text{) then } 0 \text{ else (} m + (\text{mult (} n - 1 \text{) m}) \text{)}
\]

\[
fac \ n = \text{if (} n == 0 \text{) then } 1 \text{ else (} \text{mult } n \ (\text{fac (} n - 1 \text{)}) \text{)}
\]

FL = Factorial Language
Abstract syntax of FL

data Function = Function Name [Name] Expr

type Name = String

data Expr =
    Literal Int
    | Argument Name
    | Binary Ops Expr Expr
    | IfThenElse Expr Expr Expr
    | Apply Name [Expr]
    | Let Name Expr Expr

data Ops = Equal | Plus | Minus
Inlining lets – an illustration

Before inlining

\[
\begin{align*}
mult\ n\ m &= \text{if } (n==0) \text{ then } 0 \text{ else } (m + (mult\ (n - 1)\ m)) \\
fac\ n &= \text{if } (n==0) \text{ then } 1 \text{ else } \text{let } y = (fac\ (n - 1)) \text{ in } (mult\ n\ y)
\end{align*}
\]

After inlining

\[
\begin{align*}
mult\ n\ m &= \text{if } (n==0) \text{ then } 0 \text{ else } (m + (mult\ (n - 1)\ m)) \\
fac\ n &= \text{if } (n==0) \text{ then } 1 \text{ else } (mult\ n\ (fac\ (n - 1)))
\end{align*}
\]
Encoding of focus

**data** Expr =
- Literal Int
- Argument Name
- Binary Ops Expr Expr
- IfThenElse Expr Expr Expr
- Apply Name [Expr]
- Let Name Expr Expr
- Focus Expr

```
fac n = if (n == 0) then 1 else <<let y = (fac (n - 1)) in (mult n y)>>
```

In an OO language/environment with "references", we would instead maintain a reference of the corresponding "object".

**Abstract syntax**

**Concrete syntax**
Inlining strategy

Iterate over all functions.

- Search for expressions with focus.
- Replace id by expression when in focus.
- Rebuild expressions otherwise.

This strategy is remarkably abstract but also imprecise.
Implementation in Haskell

- `Parser.hs` – omitted here
- `PrettyPrinter.hs` – omitted here
- `Evaluator.hs` – briefly discussed
- `Inline.hs` – clearly of interest

Refactoring must preserve semantics.
Evaluator.hs

```haskell
module Evaluator where

import Prelude hiding (lookup)
import Data.Map
import AbstractSyntax

eval :: [Function] -> Expr -> Maybe Int
eval fs e = eval empty e
  where
    eval :: Map Name Int -> Expr -> Maybe Int
    eval m (Literal i) = Just i
    eval m (Argument n) = lookup n m
    eval m (Binary o x y) = do
      x' <- eval m x
      y' <- eval m y
      return (case o of
        Equal -> if x' == y' then -1 else 0
        Plus     -> x' + y'
        Minus    -> x' - y')
    eval m (IfThenElse x y z) = do
      x' <- eval m x
      if x' /= 0 then eval m y else eval m z
```

Use a map for the binding of parameters and let variables.
Otherwise, simply recurse into expressions “as usual”. We use “Maybe” to be prepared for failure of evaluation.

Evaluator.hs cont’d

eval m (Apply n es) = do
  is <- mapM (eval m) es
  (ns,e) <- lookupFunction
  let m' = fromList (zip ns is)
  eval m' e
where
lookupFunction :: Maybe ([Name], Expr)
lookupFunction
  = head ( [ Just (ns,e) | (Function n' ns e) <- fs, n == n' ] ++ [Nothing] )
eval \( m \) (Let \( n \) \( x \) \( y \)) = do
  \( x' \) <- eval \( m \) \( x \)
  let \( m' = \) insert \( n \) \( x' \) \( m \)
  eval \( m' \) \( y \)

The environment (the “map”) is updated to assume the value of \( x \) for \( n \) within the expression evaluation of \( y \).
module Inline

import AbstractSyntax

-- Iterate over all functions

inline :: [Function] -> Maybe [Function]
inline = mapM inline
  where
    inline (Function n ns e)
      = copy e >>= Just . Function n ns

Overall idea: start in “copy” mode while processing a function’s body. Switch to “replace” mode when hitting a let in focus.
```
module Inline where

import AbstractSyntax -- Iterate over all functions

inline :: [Function] -> Maybe [Function]
inline = mapM inline

inline (Function n ns e) =
  copy e >>= Just . Function n ns

-- Copy mode of inlining

copy :: Expr -> Maybe Expr

copy @(Literal _) = Just x

copy @(Argument _) = Just x

copy (Binary o x y) = do
  x' <- copy x
  y' <- copy y
  Just (Binary o x' y')

copy (IfThenElse x y z) = do
  x' <- copy x
  y' <- copy y
  z' <- copy z
  Just (IfThenElse x' y' z')

copy (Apply n es) = do
  es' <- mapM copy es
  Just (Apply n es')
```

This is entirely regular code for recursive copying. We drill into terms until we hit a let in focus.

“Strategic programming” can be used to reduce the amount of boilerplate code at hand.
Copy (Let n x y) = do
  x' <- copy x
  y' <- copy y
  Just (Let n x' y')

Copy (Focus (Let n x y)) = replace n x y

Copy (Focus x) = copy x

Switch to "replace" mode as we hit the focus also assuming that the focus surrounds a let as opposed to some other language construct.
-- Replace mode of inlining

```haskell
replace :: Name -> Expr -> Expr -> Maybe Expr
replace n e = replace'
  where
    replace' x@(Literal _) = Just x
    replace' (Argument n') = Just (if n==n'
      then e
      else Argument n')
```

Copy literal
If the name at hand is the one from the focused let, then replace accordingly, otherwise copy.
Inline.hs cont’d

```haskell
replace' (Binary o x y) = do
  x' <- replace' x
  y' <- replace' y
  Just (Binary o x' y')

replace' (IfThenElse x y z) = do
  x' <- replace' x
  y' <- replace' y
  z' <- replace' z
  Just (IfThenElse x' y' z')

replace' (Apply n es) = do
  es' <- mapM replace' es
  Just (Apply n es')

replace' (Let n x y) = do
  x' <- replace' x
  y' <- replace' y
  Just (Let n x' y')
```

Boilerplate code
replace' (Focus _)
  = Nothing -- no support for nested foci

What to do when hitting a focus inside a focus? One option is to make the transformation fail. What other options do we have?
The underlying code leverages the Strafunski StrategyLib for Haskell.

http://www.haskell.org/haskellwiki/Applications_and_libraries/Generic_programming/Strafunski


Language-parametric refactoring

https://github.com/slecours/slecours/tree/master/sources/refactoring/joos/rule02
### Variations on generic extraction

<table>
<thead>
<tr>
<th>Paradigm</th>
<th>Focus</th>
<th>Abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>OO programming</td>
<td>statements</td>
<td>method</td>
</tr>
<tr>
<td>OO programming</td>
<td>features</td>
<td>class</td>
</tr>
<tr>
<td>Functional programming</td>
<td>expression</td>
<td>function</td>
</tr>
<tr>
<td>Functional programming</td>
<td>type expression</td>
<td>datatype</td>
</tr>
<tr>
<td>Functional programming</td>
<td>functions</td>
<td>type class</td>
</tr>
<tr>
<td>Functional programming</td>
<td>literal</td>
<td>predicate</td>
</tr>
<tr>
<td>Logic programming</td>
<td>EBNF phrase</td>
<td>nonterminal</td>
</tr>
<tr>
<td>Syntax definition</td>
<td>code fragment</td>
<td>macro</td>
</tr>
<tr>
<td>Preprocessing</td>
<td>content particle</td>
<td>element type</td>
</tr>
<tr>
<td>Document processing</td>
<td>sentences</td>
<td>paragraph</td>
</tr>
<tr>
<td>Cobol programming</td>
<td>sentences</td>
<td>subprogram</td>
</tr>
<tr>
<td>Cobol programming</td>
<td>data description entries</td>
<td>copy book</td>
</tr>
<tr>
<td>Cobol programming</td>
<td>data description entries</td>
<td>group field</td>
</tr>
<tr>
<td>Cobol programming</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extraction of a Haskell datatype

Input program with focus

\[
\begin{align*}
\textbf{data} & \quad \textbf{Prog} \quad = \quad \textbf{Prog} \ \textbf{ProgName} [[\textbf{Dec}]] [[\textbf{Stat}]] \\
\textbf{data} & \quad \textbf{Dec} \quad = \quad \textbf{VDec} \ \textbf{Id} \ \textbf{Type} \ | \ ... \\
\textbf{data} & \quad \textbf{Stat} \quad = \quad \textbf{Assign} \ \textbf{Id} \ \textbf{Expr} \ | \ \textbf{If} \ \textbf{Expr} \ \textbf{Stat} \ \textbf{Stat} \ | \ ... \\
\end{align*}
\]

Output program after extraction and integration

\[
\begin{align*}
\textbf{data} & \quad \textbf{Prog} \quad = \quad \textbf{Prog} \ \textbf{ProgName} \ \textbf{Block} \\
\textbf{data} & \quad \textbf{Block} \quad = \quad \textbf{Block} [[\textbf{Dec}]] [[\textbf{Stat}]] \\
\textbf{data} & \quad \textbf{Dec} \quad = \quad ... \\
\textbf{data} & \quad \textbf{Stat} \quad = \quad ... \ | \ \textbf{BlockStat} \ \textbf{Block} \\
... \\
\end{align*}
\]
Some other refactorings

- Inlining (inverse of extraction)
- Introduction / elimination
- Fold / unfold (similar extract / inline)
- Pull up / push down
- Add parameter
- Reorder
- ...

So much refactoring. So little time.
Generic extraction
THE IDEA

See also the paper “Towards Generic Refactoring” by Ralf Lämmel.
http://doi.acm.org/10.1145/570186.570188
Steps of generic extraction

1. Lookup focused fragment.
2. Determine free names in focused fragment.
3. Enforce language-dependent check on focus.
5. Find host for new abstraction.
6. Introduce abstraction.
7. Construct application.
8. Replace focus by application.
Lookup focused fragment
in a language-parametric manner

A generic piece of refactoring functionality

\[
\text{selectFocus} :: (\text{Term } f, \text{Term } t) \\
\Rightarrow (f \rightarrow \text{Maybe } f) \quad \text{-- Get focus} \\
\rightarrow t \quad \text{-- Input term} \\
\rightarrow \text{Maybe } f \quad \text{-- Focused term}
\]

\[
\text{selectFocus unwrap} = \text{adhoc} (\text{const Nothing}) \text{unwrap} \\
\quad \text{‘choice’} \text{oneTU selectFocus}
\]

A language-specific instantiation

\[
\text{selectStatement} :: \text{Term } t \Rightarrow t \rightarrow \text{Maybe Statement} \\
\text{selectStatement} = \text{selectFocus unwrapStatement} \\
\text{where} \\
\quad \text{unwrapStatement} (\text{StatementFocus } \text{stat}) = \text{Just } \text{stat} \\
\quad \text{unwrapStatement } _- = \text{Nothing}
\]
More aspects that deserve a language-parametric approach

- Generic rewriting in a focus / scope
- Generic marking of an enclosing scope
- Generic free name analyses
- Generic abstract syntax
Generic rewriting in a focus / scope

\[
\text{replaceFocus :: } (\text{Term } t, \text{Term } t') \\
\Rightarrow (t \rightarrow \text{Maybe } t) \quad -- \text{Transform focus} \\
\rightarrow t' \quad -- \text{Input term} \\
\rightarrow \text{Maybe } t' \quad -- \text{Output term}
\]

\[
\text{replaceFocus trafo = once_td (adhocTP fail trafo)}
\]
markHost :: (Term f, Term h, Term t) → (f → Bool) -- Test focus
    → (h → h) -- Wrap host
    → t -- Input term
    → Maybe t -- Output term
markHost testFocus wrapHost = host ‘above’ focus

where
host = adhocTP fail (Just ° wrapHost)
focus = adhocTU fail (guard ° testFocus)
Generic free name analysis

\[
gfreeNames :: (\text{Eq } n, \text{Term } \alpha) \\
\Rightarrow (\forall \alpha. \text{Term } \alpha \Rightarrow \alpha \rightarrow [n]) \\
\rightarrow (\forall \alpha. \text{Term } \alpha \Rightarrow \alpha \rightarrow [n]) \\
\rightarrow \alpha \\
\rightarrow [n]
\]

-- Identify declarations
-- Identify references
-- Input term
-- Free names

\[
gfreeNames \text{ declared referenced } x = \\
((\text{referenced } x) \\
\text{‘union‘} \\
(\text{allTU union } [] (gfreeNames \text{ declared referenced})) x \\
) \text{ \textbackslash\textbackslash declared } x
\]
class

Syntactical domains

(  
  Term abstr,  -- Term type for abstraction
  Eq name,    -- Names of abstractions
  Term [ abstr],  -- Lists of abstractions
  Term apply   -- Term type for applications
 )

⇒ Abstraction abstr name tpe apply
Dependencies between syntactical domains

|    | \textit{abstr} \rightarrow \textit{name},  \\
|    | \textit{abstr} \rightarrow \textit{tpe},    \\
|    | \textit{abstr} \rightarrow \textit{apply},  \\
|    | \textit{apply} \rightarrow \textit{name},   \\
|    | \textit{apply} \rightarrow \textit{abstr}   |
where

Observers

\[
\begin{align*}
\text{getAbstrName} & \quad :: \text{abstr} \rightarrow \text{Maybe name} \\
\text{getAbstrParas} & \quad :: \text{abstr} \rightarrow \text{Maybe} \left[ (\text{name}, \text{tpe}) \right] \\
\text{getAbstrBody} & \quad :: \text{abstr} \rightarrow \text{Maybe apply} \\
\text{getApplyName} & \quad :: \text{apply} \rightarrow \text{Maybe name} \\
\text{getApplyParas} & \quad :: \text{apply} \rightarrow \text{Maybe} \left[ (\text{name}, \text{tpe}) \right]
\end{align*}
\]

Constructors

\[
\begin{align*}
\text{constrAbstr} & \quad :: \text{name} \rightarrow \left[ (\text{name}, \text{tpe}) \right] \rightarrow \text{apply} \rightarrow \text{Maybe abstr} \\
\text{constrApply} & \quad :: \text{name} \rightarrow \left[ (\text{name}, \text{tpe}) \right] \rightarrow \text{Maybe apply}
\end{align*}
\]
Generic extraction, finally

extract :: (Term prog, Abstraction abstr name tpe apply)
⇒ (∀α. Term α ⇒ α → [(name, tpe)])
→ (∀α. Term α ⇒ α → [name])
→ (apply → Maybe apply)
→ ([abstr] → [abstr])
→ ([abstr] → Maybe [abstr])
→ (((name, tpe]) → apply → Bool)
→ name
→ prog
→ Maybe prog

-- Identify declarations
-- Identify references
-- Unwrap focus
-- Wrap host
-- Unwrap host
-- Check focus
-- Name for abstraction
-- Input program
-- Output program
The definition of generic extraction

\[
\text{extract declared referenced unwrap wrap unwrap'} \text{ check name prog} = \text{do}
\]

\[
\text{-- Operate on focus}
\]

\[
(bound, focus) \leftarrow \text{boundTypedNames declared unwrap prog}
\]

\[
\text{free} \leftarrow \text{return (freeTypedNames declared referenced bound focus)}
\]

\[
\text{guard (check bound focus)}
\]

\[
\text{-- Construct abstraction}
\]

\[
\text{abstr} \leftarrow \text{constrAbstr name free focus}
\]

\[
\text{-- Insert abstraction}
\]

\[
\text{prog'} \leftarrow \text{markHost (maybe False (const True) \circ \text{unwrap}) wrap prog}
\]

\[
\text{prog''} \leftarrow \text{introduce declared referenced unwrap' abstr prog'}
\]

\[
\text{-- Construct application}
\]

\[
\text{apply} \leftarrow \text{constrApply name free}
\]

\[
\text{-- Replace focus by application}
\]

\[
\text{replaceFocus (maybe Nothing (const (Just \text{apply}) \circ \text{unwrap}) \text{prog}'')}
\]
Framework instantiation

- Ingredients of generic algorithms
- Instantiation of Abstraction class
- Language-specific checks

Exemplified for Java subset JOOS
Focus processing in JOOS

Syntax extensions

\[\textbf{data} \; \text{Statement} = \cdots \; | \; \text{StatementFocus} \; \text{Statement} \]

\[\textbf{data} \; \text{MethodDecl} = \cdots \; | \; \text{MethodDeclFocus} \; [\; \text{MethodDecl} \;] \]

Focus on statements and lists of method declarations

\[\text{wrapStatement} = \text{StatementFocus} \]

\[\text{unwrapStatement} \; (\text{StatementFocus} \; x) = \text{return} \; x \]

\[\text{unwrapStatement} \; _ = \text{mzero} \]

\[\text{wrapMethods} \; xs = [\; \text{MethodDeclFocus} \; xs \;] \]

\[\text{unwrapMethods} \; [\; \text{MethodDeclFocus} \; xs \;] = \text{return} \; xs \]

\[\text{unwrapMethods} \; _ = \text{mzero} \]
Free-name analysis for JOOS

Datatype for kinds of relevant JOOS identifiers

\[
data TypeJoos \equiv ExprType \ Type
\]

Declared names (with type)

\[
declaredJoos :: TU \ [(\text{Identifier}, TypeJoos)] \ \text{Identity}
declaredJoos \equiv \ \text{adhocTU} (\ \text{adhocTU} (constTU [[]])
\ (\text{Identity} \circ \text{declaredBlock}))
\ (\text{Identity} \circ \text{declaredMeth})
\]

where

\[
declaredBlock (\text{Block vds \_})
\equiv \text{map} (\lambda (\text{VarDecl t i}) \to (i, ExprType t)) \ vds
\]

\[
declaredMeth (\text{MethodDecl \_ \_ (Formals fps) \_})
\equiv \text{map} (\lambda (\text{Formal t i}) \to (i, ExprType t)) \ fps
\]

Further ingredients omitted
Instance of Abstraction class for JOOS

instance Abstraction
  MethodDecl
  Identifier
  TypeJoos
  Statement
  -- abstr
  -- name
  -- tpe
  -- apply
where

-- Observers

getAbstrName (MethodDecl _ i _ _) = Just i
getAbstrParas (MethodDecl _ _ (Formals fps) _) 
  = Just (map (λ(Formal t i) → (i, ExprType t)) fps)
getAbstrBody (MethodDecl _ _ _ b) 
  = Just (BlockStat b)
...

-- Constructors

\[
\text{constrAbstr } n \; l \; a \\
= \text{maybe Nothing} \\
(\lambda fps \rightarrow \text{Just (MethodDecl Nothing } n \\
(\text{Formals } fps) \\
(\text{toBlock } a))) \\
(\text{mapM } \text{toFormal } l)
\]

where

...
Language-specific transformations by parameter passing

Type of transformation on JOOS programs

\[
\text{type } \text{TrafoJoos} = \text{Program} \rightarrow \text{Maybe Program}
\]

Extraction of a statement to constitute a new method declaration

\[
\text{extractJoos} :: \text{Identifier} \rightarrow \text{TrafoJoos}
\]

\[
\text{extractJoos} = \text{extract}
\]

\[
\begin{align*}
\text{extractJoos} &= \text{extract} \\
&= \text{extract} \\
&= \text{extract} \\
&= \text{extract} \\
&= \text{extract} \\
&= \text{extract} \\
&= \text{extract} \\
&= \text{extract}
\end{align*}
\]

\[
\begin{align*}
\text{extractJoos} &= \text{extract} \\
&= \text{extract} \\
&= \text{extract} \\
&= \text{extract} \\
&= \text{extract} \\
&= \text{extract} \\
&= \text{extract} \\
&= \text{extract}
\end{align*}
\]

\[
\begin{align*}
\text{extractJoos} &= \text{extract} \\
&= \text{extract} \\
&= \text{extract} \\
&= \text{extract} \\
&= \text{extract} \\
&= \text{extract} \\
&= \text{extract} \\
&= \text{extract}
\end{align*}
\]

\[
\begin{align*}
\text{extractJoos} &= \text{extract} \\
&= \text{extract} \\
&= \text{extract} \\
&= \text{extract} \\
&= \text{extract} \\
&= \text{extract} \\
&= \text{extract} \\
&= \text{extract}
\end{align*}
\]

\[
\begin{align*}
\text{extractJoos} &= \text{extract} \\
&= \text{extract} \\
&= \text{extract} \\
&= \text{extract} \\
&= \text{extract} \\
&= \text{extract} \\
&= \text{extract} \\
&= \text{extract}
\end{align*}
\]

\[
\begin{align*}
\text{extractJoos} &= \text{extract} \\
&= \text{extract} \\
&= \text{extract} \\
&= \text{extract} \\
&= \text{extract} \\
&= \text{extract} \\
&= \text{extract} \\
&= \text{extract}
\end{align*}
\]

\[
\begin{align*}
\text{extractJoos} &= \text{extract} \\
&= \text{extract} \\
&= \text{extract} \\
&= \text{extract} \\
&= \text{extract} \\
&= \text{extract} \\
&= \text{extract} \\
&= \text{extract}
\end{align*}
\]

\[
\begin{align*}
\text{extractJoos} &= \text{extract} \\
&= \text{extract} \\
&= \text{extract} \\
&= \text{extract} \\
&= \text{extract} \\
&= \text{extract} \\
&= \text{extract} \\
&= \text{extract}
\end{align*}
\]
where

\[
\text{check } _f = \text{and} \ [\text{noReturns } f, \text{noFrees } f]
\]

\[
\text{noReturns} = \text{maybe True (const False)} \circ \text{applyTU (onetd (adhocTU fail (}\lambda s \to \text{case } s \text{ of} \\
\text{ReturnStat } _- \to \text{Just } () \\
_- \to \text{Nothing}))})
\]

\[
\text{noFrees} = (\equiv) \ [\ ] \circ \text{freeNames declaredJoos definedJoos}
\]
Concluding remarks

- Refactoring is amenable to automated program transformation.
- Refactoring is of great importance in software development.
- Refactoring quickly gets complicated (interesting):
  - Non-trivial preconditions must be checked.
  - Non-trivial analyses are needed for other reasons.
  - Conservative approaches quickly lead to code bloat.
- Haskell is a lot of fun for studying this domain.