Strafunski

*Functional Strategy Combinators*

http://www.cs.vu.nl/Strafunski/

Joost Visser (Universidade do Minho)
joint work with Ralf Lämmel (VU & CWI)
Using FP for language processing applications, e.g.:
- Program analysis and reverse engineering
- Refactoring and re-engineering

Reuse external components for e.g.:
- Parsing
- Graph visualization

Employ generic traversal for
- Conciseness (focus on relevant data constructors)
- Robustness (isolate against data structure change)
Strafunski

What is it?

Strafunski = Strategies + functions

- a Haskell-based bundle
- for generic programming, based on the concept of a functional strategy, and
- for language processing, using GLR

Strafunski = combinator library + precompiler + parser generator
Strafunski

Outline of this talk

- Concepts
- Design patterns
- Tools
- Applications
Strafunski

Traversal schemes

Recursion scheme

One-step traversal

Recursive call

```
topdown s = s `seq` (all (topdown s))
bottomup s = (all (bottomup s)) `seq` s
once_td s = s `choice` (one (once_td s))
...
What are functional strategies?

Characteristics:
- first-class, generic functions
- composed and updated in combinator style
- allow generic traversal into subterms
- mix type-specific and uniform behaviour

functional strategy $\neq$ parametric polymorphic function
functional strategy $\neq$ polytypic function

composed from simple combinators

induction over sums-and-products

freely mix generic and type-specific behaviour

only uniform behaviour
Strafunski
What are functional strategies?

**Example:** increment all integers in a term

```haskell
increment = apply (topdown (adhoc identity (+1)))
```

> :i increment
increment :: Term a => a -> a
> increment [0,1,2]
[1,2,3]
> increment (True,[0,1],Just 2)
(True,[1,2],Just 3)
**Strafunski**

*What is it good for?*

**Example:**

- Haskell itself (30 datatypes, 100 constructors)
- Collect all type constructor names

```haskell
def refTypeNames :: Term a => a -> [HsName]
def refTypeNames = runId . applyTU traversal
where
    traversal = crush nodeAction
    nodeAction = adhocTU (constTU []) getName
    getName (HsTyCon (UnQual n)) = return [n]
    getName _ = return []
```

- Mention two constructors only
- Works on any Haskell fragment / dialect
Strafunski

vs. Scrap your boilerplate

Scope:
- Data.Generics: basic strategy combinators
- Strafunski: basic combinators + library + tools

Availability:
- Data.Generics: available in GHC version 6.2
- Strafunski: works with GHC and Hugs and NHC

Names:
- Data.Generics: Data, extM/Q, gmapM/Q,...
- Strafunski: Term, adhocTP/TU, allTP/TU,...

Future:
- Strafunski uses Data.Generics as basis (optionally)
Strafunski

Library themes

- Traversal (full_td, once_td, stop_td, ...)
- Fixpoint (outermost, innermost, ...)
- Path (below, above,...)
- Name (freeNames,...)
- Keyhole (selectFocus, replaceFocus, deleteFocus)
- Metrics (typeMetric, predMetric, depthWith, ...)
Strafunski

Outline of this talk

- Concepts
- **Design patterns**
- Tools
- Applications
Strafunski

Design patterns

Keyhole Operation

Rewrite Step

Local Effect

Parameter for

Type-specific ingredient for

Generic Rewrite Step

Success by Failure

Localize by

Partiality for branches in

Wire-up node actions within

Traversals Scheme

Traversals

Traversal

Failure

Success

Assign roles to

Instantiate to

Role Play

Meta Scheme

Circuitry

Propagation

Provide interface to

Instantiate to
Inten: Capture a single type-specific computation step

Motivation: By capturing type-specific computations and naming them, they can easily be reused in different contexts.

Schema:

```plaintext
step  :: T -> T'
step pat = rhs
step v = ...
```

Sample code:

```plaintext
refTypes :: HsType -> [HsName]
refTypes (HsTyCon (UnQual n)) = [n]
refTypes _ = [ ]
```
Strafunski

Generic Rewrite Step

**Intent:** Lift type-specific rewrite steps to all types

**Motivation:** At some point in the synthesis of generic programs, type-specific steps must be made generic.

**Schema:**

```
generic = default `adhoc` step1 `adhoc` step2
```

**Sample code:**

```
anyTypes :: TU [HsName] Identity
anyTypes = constTU [ ]
    `adhocTU` (return . decTypes)
    `adhocTU` (return . refTypes)
```
**Intent:** Instantiate a traversal scheme with generic rewrite steps.

**Motivation:** You can construct your own traversal by instantiating a predefined traversal scheme e.g. from Strafunski's library.

**Schema:**
\[
\text{instantiation} = \text{scheme arg1 ... argN}
\]

**Sample code:**
```
allTypes :: TU [HsName] Identity
allTypes = crush anyTypes
```

Using the predefined combinator:
```
crush :: (Monad m, Monoid u) => TU u m -> TU u m
```
Strafunski

Keyhole Operation

Aka: Wrapper Worker

Intent: Do not expose strategy type to the top level.

Motivation: On the inside, you can work with the full power of strategies, while on the outside, all you see is a plain function without any trace of TP, TU, Term.

Schema:

\[
\text{wrapper fp1 ... fpN = ... apply worker ...}
\]
\[
\text{where worker = ... `adhoc` ...}
\]

Sample code:

\[
\text{isFreshType :: HsName -> HsModule -> Bool}
\]
\[
\text{isFreshType n = runIdentity . applyTU worker}
\]
\[
\text{where worker = allTypes `before` isNotElem}
\]
\[
\text{isNotElem = not . (elem n)}
\]
Strafunski

Generic Container

**Intent**: Use a strategy as a generic data container.

**Motivation**: Terms of different types sometimes need to be stored in the same container.

**Sample code**:

```haskell
  type GSet = TU () Maybe
  emptyGS = failTU
  fullGS = constTU mempty
  elemGS e s = maybe False (const True) (applyTU s e)
  addGS e s = modifyTU s e (return mempty)
  rmGS e s = modifyTU s t mzero

  modifyTU f e = adhocTU f . modify (applyTU f) t
  modify f x y = \x' -> if x == x' then y else f x'
```
Strafunski

Outline of this talk

• Concepts
• Design patterns
• Tools
• Applications
Strafunski

What makes it work?

- no new language (cf. PolyP, GH, FISh)
- rely on Term class that captures extras
- instantiate for every algebraic datatype
- use precompiler (extended version of DrIFT)
- or add derive Data,Typeable to all your datatypes (with GHC 6.2).
Strafunski Architecture

Library

- StrategyLib
  - Fixpoint Theme
  - Path Theme
  - Name Theme
  - StrategyPrimitives
  - TermRep

Application

- Strategies
- Datatypes
  - Term instances

Precompiler

Strafunski = Library + Precompiler
Source code:
- SDF to specify grammar
- SGLR to parse
- ATerms to exchange ASTs

Documents:
- DTD to specify document structure
- XML to exchange documents
- HaXML to read / write
Getting to terms

- DTD
- DrIFT
- Term instances
- XML instances
- A Term instances
- Dtd2Haskell
- Sdf2Haskell
- PGEND
- GLR parser
- Program
- AST
- Strategies
- Document
- Datatypes

Strafunski
Strafunski

Strafunski:
• StrategyLib
• ATermLib
• ( DrIFT-Strafunski )
• Sdf2Haskell

Uses:
• Haskell compiler / interpreter (GHC / Hugs) and Haskell libraries
• parser & parse table generator (SGLR & PGEN)
Strafunski

Outline of this talk

• Concepts
• Design patterns
• Tools
• Applications
Java metrics and reverse engineering.
- SDF grammar for Java
- E.g. count conditionals, nesting depth, ...
- E.g. Extract conditional call graph

Java refactoring.
- E.g. Extract Method refactoring

Meta-lang = object-lang = Haskell.
- Use same parser as Haddock
- E.g. do elimination, newtype introduction

Cobol reverse engineering.
- SDF grammar for Cobol
- Extract perform graph
Principles:

Typed combinators for generic traversal (PADL 2002)

Applications:

A Strafunski application letter (PADL 2003)

Cook book:

Design patterns for functional strategic programming (RULE 2002)

Implementation:

Strategic polymorphism requires just two combinators! (IFL 2002)

http://www.cs.vu.nl/Strafunski/