Advanced Functional Programming

LMCC & LESI, Universidade do Minho

Strafunski: Exercises

1 Preliminaries

The subdirectory `StrategyLib` contains (version 4.0 of) Strafunski's strategy library. We assume you have either Hugs or GHC installed.

2 Your first strategic program

To get acquainted with the basics of strategic programming you will construct your first strategic program in a few simple steps.

2.1 Load the library

Exercise 2.1 *Create the file `MyFirstStrategicProgram.hs` with the following initial content:*

```
module MyFirstStrategicProgram where
import StrategyLib
```

Load this program into Hugs or GHCi as follows:

```
ghci -fglasgow-exts -fallow-overlapping-instances \
     -fallow-undecidable-instances -package data \
 -iStrategyLib-4.0/library:StrategyLib-4.0/models/drift-default: \
 MyFirstStrategicProgram
```

```
hugs +o -98 +N  \
     -PStrategyLib-4.0/library:StrategyLib-4.0/models/drift-default: \
     MyFirstStrategicProgram
```

If all is well, the strategy library will be loaded into the interpreter, ready for use. With the `:t` command, you can ask the interpreter for the types of the library's combinators. Try this for `idTP`, `applyTP`, `adhocTP`, and `topdown`.

```
MyFirstStrategicProgram> :t idTP
idTP :: ...
MyFirstStrategicProgram> :t applyTP
applyTP :: ...
MyFirstStrategicProgram> :t adhocTP
adhocTP :: ...
MyFirstStrategicProgram> :t topdown
topdown :: ...
```

You will use these combinator in the next exercise.

In these types, you see occurances of the type constructor `TP`, and the type class `Term`. The constraint `Term a` basically means: strategic functionality is activated for type `a`. If you are using Hugs, the command `:i Term` will reveal that this is the case for most of Haskell's predefined types. The type `TP m`, where `m` is a monad, is the type of (monadic) strategic functions that takes a term of any type `x` into a result of type `x`, i.e. it is a *type-preserving generic function*.

2.2 A type-preserving strategic function

Next, you will construct a simple type-preserving strategic function using the combinators mentioned above.

Exercise 2.2 *Add the following import and function declaration to your first strategic program:*

```
import MonadIdentity

increment :: Term a => a -> a
increment = runIdentity . applyTP strategy
  where
    strategy = topdown (adhocTP idTP inc)
    inc x    = return (x + 1::Int)
```

As the type indicates, the function `increment` *works on terms of any type, and maps them to a result of the same type. Apply it to terms of several types, and inspect the results:*

```
MyFirstStrategicProgram> increment True
...
MyFirstStrategicProgram> increment (42::Int)
...
MyFirstStrategicProgram> increment [42::Int]
...
MyFirstStrategicProgram> increment ([42::Int],True,Just (7::Int))
...
```

As you can see, `increment` returns its argument unchanged, except that 1 has been added to any `Int` that occurs in it. The shape of the term, and the place where the `Int`s occur in it, are not important, and remain unchanged.

Note that, if `increment` had been an ordinary parametric polymorphic function of type `a -> a`, it could only have been the identity function. But the constraint `Term a` indicates that `increment` is not paramteric polymorphic, but that it is a *strategic function*, which exhibits a mixture of generic and type-specifc behaviour. The generic behaviour is the identity, as expressed with the `idTP` combinator. The type specific behaviour is adding 1, as expressed by the local `inc` function. The `adhocTP` combinator combines these two into a single generic function with mixed behaviour. Finally, the `topdown` combinator applies this mixed behaviour to all subterms in a top-down fashion.

2.3 Traversal variations

The `topdown` combinator is a synonym for the `full_tdTP` combinator. As the latter name indicates, it performs a *full* traversal, i.e. it visits every subterm, in a top-down fashion.

Exercise 2.3 *Inspect the types of the following combinators:*

```
MyFirstStrategicProgram> :t full_tdTP
full_tdTP :: ...
MyFirstStrategicProgram> :t full_buTP
full_buTP :: ...
MyFirstStrategicProgram> :t stop_tdTP
stop_tdTP :: (MonadPlus m) => ...
MyFirstStrategicProgram> :t once_tdTP
once_tdTP :: ...
MyFirstStrategicProgram> :t once_buTP
once_buTP :: ...
```

Note the occurrance of `MonadPlus` in the context of some of these combinators. An example of `MonadPlus` is the `Maybe` type constructor. It is used to indicate that a strategic function succeeds (`Just a`) or fails (`Nothing`).

To see these traversal variations in action, we will use a tree datatype `Tree` defined in the module `Tree` as:

```
data Tree = Tree Int [Tree]
```

Add an `import Tree` to your first strategic program, as well as the following function:

```
flip :: Term a => (TP Maybe -> TP Maybe) -> a -> a
flip traversal a = maybe a id (applyTP strategy a)
  where
    strategy = traversal (adhocTP failTP flp)
    flp (Tree i []) = mzero
    flp (Tree i ts) = return (Tree i (reverse ts))
```

The `flip` function takes as first argument the `traversal` combinator to be applied. The type-specific behaviour defined by the local `flp` function is the reversal of the list of subtrees. The strategy combinator `failTP` is the dual of `idTP`: it fails to produce a result for any input term.

Now, apply flip to a tree, using various traversal combinators as arguments:

```
MyFirstStrategicProgram> flip once_tdTP myTree
...
MyFirstStrategicProgram> flip once_buTP myTree
...
MyFirstStrategicProgram> flip stop_tdTP myTree
...
MyFirstStrategicProgram> flip full_tdTP myTree
Tree 1 [Tree 2 [Tree 3 [],Tree 4 []],Tree 5 [Tree 6 [],Tree 7 []]]
MyFirstStrategicProgram> flip (full_tdTP.tryTP) myTree
...
MyFirstStrategicProgram> flip (full_buTP.tryTP) myTree
...
```

For example, define `myTree` as follows:

```
myTree = Tree 1 [Tree 2 [Tree 3 [],Tree 4 []],Tree 5 [Tree 6 [],Tree 7 []]]
```

Looking at the results, you should be able to understand the behaviour of each traversal combinator. Consult the online Haddock documentation of Strafunski (TraversalTheme) to verify your understanding.

Note the use of `tryTP`. This combinator attempts to apply its argument strategy. If this fails, it does not propagate the failure, but returns the input term. We use `tryTP` in combination with the full traversals, since these need their argument strategies to succeed on every term to succeed themselves.

2.4 A type-unifying strategic function

So far, we have seen only one side of Strafunski: type-preserving strategies. To complement these, the library also offers type-unifying strategies.

Exercise 2.4 *Inspect the types of the following combinators:*

```
MyFirstStrategicProgram> :t constTU
constTU :: ...
MyFirstStrategicProgram> :t applyTU
applyTU :: ...
MyFirstStrategicProgram> :t adhocTU
adhocTU :: ...
MyFirstStrategicProgram> :t collect
collect :: ...
MyFirstStrategicProgram> :t crush
crush :: ...
MyFirstStrategicProgram> :t full_tdTU
full_tdTU :: ...
MyFirstStrategicProgram> :t once_tdTU
once_tdTU :: ...
MyFirstStrategicProgram> :t once_buTU
once_buTU :: ...
```

Consult the online Haddock documentation of Strafunski to find out what these combinators do.

The type `TU a m`, where `m` is a monad, is the type of (monadic) strategic functions that takes a term of any type `x` into a result of the designated type `a`, i.e. it is a *type-unifying generic function*. While type-preserving strategies perform *transformations* of terms, type-unifying strategies are used to program `queries` of terms.

Exercise 2.5 *Add the following function definition to your first strategic program:*

```
list :: Term a => a -> [Int]
list = runIdentity . applyTU strategy
  where
    strategy = collect (adhocTU (constTU []) lst)
    lst x    = return [x]
```

As you can see, this function employs type-unifying strategy combinators. The type-specific behaviour defined by the local function `lst` is to put an integer into a singleton list. The generic behaviour of `constTU []` is to return an empty list. The collect strategy takes care of traversing the term in bottom-up fashion and appending all the singleton and empty lists into a final result.

Try the `list` function on various terms of various types. Create variations on `list` that employ different type-unifying traversal strategies, and test them on various terms.

So far, we have applied our strategies on terms built from Haskell's predefined types and the simple user-defined `Tree` type. But the power of strategic programming is best experienced when processing terms build from large sets of mutually recursive datatypes. Examples of these are language syntaxes and file formats. The following sections demonstrate this on the abstract syntax of Haskell itself, and on XML document trees.

3 Refactoring of Haskell Programs

One of the examples included in the Strafunski distribution is a program `HsTransform` that transforms haskell source code. It performs two simple transformations: the elimination of the do constructor and the introduction of a new type definition. You can run the program (in hugs) as follows:

```
runhugs  +o -98 +N
 -P<StrategyLib>/examples/haskell:<StrategyLib>/library:<StrategyLib>/models/drift-default:
 HsTransform <inputfile> <outputfile>
```

where `StrategyLib` is the location where you installed Strafunski's StrategyLib (version 4.0-beta), and `inputfile` and `outputfile` are the names of the Haskell program to be transformed, and the file where you want the transformed program to be written to.

Exercise 3.1 *Write a little test program that makes use of various* `do` *constructions. Run the transformation program on your test program and inspect the resulting program. Run your test program as well as the transformed program to see if they behave the same.*

Note: Be sure to test nested do-expression and monadic let-expressions inside do's.

For instance, consider the following example Haskell program:

```
Put your input test module here
```

Running the `HsTransform` program on it produces the following output:

```
Put the transformed module here
```

The elimination of `do` expressions is defined in the Haskell Report, in section 3.14. This transformation is expressed in Strafunski as follows:

```
doElim     :: (Term a, MonadPlus m) => a -> m a
doElim h   =  applyTP (innermost (monoTP doElim1)) h

doElim1    :: MonadPlus m => HsExp -> m HsExp
doElim1 (HsDo [HsQualifier e])
  = return e
doElim1 (HsDo (HsQualifier e:stmts))
  = return (HsInfixApp e (HsVar (hsSymbol ">>")) (HsDo stmts))
doElim1 (HsDo (HsGenerator p e:stmts))
  = do ok <- new_name
       return (letPattern ok p e stmts)
doElim1 (HsDo (HsLetStmt decls:stmts))
  = return (HsLet decls (HsDo stmts))
doElim1 _
  = mzero
```

The function `doElim1` implements a single `do`-elimination rewrite step. It follows the definition in the Haskell Report quite closely, except that instead of Haskell's concrete surface syntax, it's abstract syntax is used.

The function `doElim` completes the rewrite step `doElim1` into a complete transformation. The `monoTP` combinator lifts the rewrite step to a *generic* rewrite step, and the `innermost` combinator applies this generic rewrite step repeatedly according to the innermost traversal strategy, until a fixpoint is reached (no more redexes remain).

Exercise 3.2 *In section 3.6 of the Haskell report a translation of the* `if_then_else` *construct is presented. Implement this translation in a strategic function named* `ifElim`*. In order to incorporate this transformation in the distributed package, just modify the Main module to import module* `IfElim`*, and modify one line in the main function into:*

```
iowrap (mpipe [doElim,data2newtypeConv,ifElim])
```

For instance, consider the following example Haskell program:

```
module TestIfElim where
f a = if (a > 1)
      then a
      else -a
```

Running the `HsTransform` program on it produces the following output:

```
module TestIfElim where
f a
  = case (a > 1) of
True -> a
False -> - a
```

This transformation is expressed in Strafunski as follows:

```
module IfElim where

import Datatypes
import DatatypesTermInstances
import StrategyLib
import Monad

ifElim     :: (Term a, MonadPlus m) => a -> m a
ifElim h   =  ...

ifElim1    :: MonadPlus m => HsExp -> m HsExp
```

Exercise 3.3 *Implement the list comprehension elimination according to the rules defined in section 3.11 of the haskell report. (if you solve this exercise please email Joost the solution to be included in the next Strafunski distribution).*

4 Querying and transforming XML Documents

Using the XML support from the HaXml project, you can easily import XML documents into Haskell, and export them from Haskell. If you can supply a DTD for your documents, you can additionally, inside Haskell, marshall between a generic representation of XML documents (verified only) and a typed Haskell representation of XML documents (validated by the Haskell type system).

As a running example, we will use documents that contain course descriptions. An example document is given in `teste.xml`, and its DTD is given in `Curso.dtd`.

To get things running you need to run `DtdToHaskell` on the given DTD to obtain a file `Curso.hs` containing the Haskell datatypes that will represent your document. Additionally, this file will contain instances of the `XmlContent` class that provides the marshalling functionality.

Exercise 4.1 *Run* `DtdToHaskell` *and inspect the content of the resulting file. Try to understand the relationship between the DTD and the Haskell types, and between XML documents and Haskell terms.*

After generating the code, we can define some wrapper functions for performing queries and transformations on given XML documents:

```
module XMLExercise where

import Curso
import Text.XML.HaXml.Parse (xmlParse)
import Text.XML.HaXml.Types (Document(..),Content(..))
import Text.XML.HaXml.Xml2Haskell (showXml,fromElem,XmlContent(..))

processQ q xmlFile
  = do xmlInput <- readFile xmlFile
       let curso = fromString xmlInput :: Curso
       result <- q curso
       putStrLn (show result)

processT t xmlFile
  = do xmlInput <- readFile xmlFile
       let curso = fromString xmlInput :: Curso
       result <- t curso
       putStrLn (showXml result)

-- Helper for parsing

fromString :: XmlContent a => String -> a
fromString s
  = forceResult (fst (fromElem [getElem (xmlParse "hoi" s)]))
    where
      getElem (Document _ _ e) = CElem e
      forceResult Nothing  = error "XML PARSE ERROR"
      forceResult (Just r) = r
```

Exercise 4.2 *Run a minimal query and a minimal transformation on the given example document* `teste.xml` *to see if everything is working properly. Use* `ghci` *as follows:*

```
ghci -cpp -fglasgow-exts -fallow-overlapping-instances\
 -fallow-undecidable-instances -package data \
  -i<StrategyLib>/library:<StrategyLib>/models/drift-default:
  -i<HaXml>/src: XMLExercise
```

where `<HaXml>` *is the place where you installed HaXml. At the prompt, try:*

```
*Main> processQ (const . putStrLn $ "Nada") "teste.xml"
*Main> processT return "teste.xml"
```

Make sure you understand what is going on. For instance, do you understand why the type annotation (`:: Curso`*) is used?*

If we make use of HaXml's marshalling functionality, writing queries and transformations on XML documents can be done by plain functional programming on the Haskell terms that represent these documents. But, since these documents can contain large numbers of different element tags and attributes, plain functional programming may quickly turn out to be clumsy.

Exercise 4.3 *Program a query that extracts all student names from a given XML document, using standard functional programming techniques. Run the query with* `processQ`.

The following code shows how we can extract student names from a given XML document using a set of recursive functions:

```
studentNames
 = processQ (return . qCurso) "teste.xml"
   where
     qCurso (Curso _ _ cadeiras)
       = ...
     qCadeira (Cadeira _ _ _ inscritos _)
       = ...
     qInscritos (Inscritos _ alunos)
       = ...
     qAluno (Aluno _ nome)
       = nome
```

Note that the recursive functions do pattern matching on all the parent types of `Aluno`. If the definition of `Curso`, `Cadeira`, or `Inscritos` is changed (for instance by allowing further elements inside them), the code above will need to be adapted as well.

Using the strategic function combinators of Strafunski, we can improve on this situation. To use Strafunski, we need to generate instances of the `Term` class from the datatypes in `Curso.hs`, using `DrIFT`.

Exercise 4.4 *Generate the file* `CursoTermInstances.hs` *with an invocation of* `DrIFT`*, as follows:*

```
> echo "module CursoTermInstances where" > CursoTermInstances.hs
> echo "import Curso" >> CursoTermInstances.hs
> echo "import TermRep" >> CursoTermInstances.hs
> DrIFT -g Term -g Typeable -r Curso.hs >> CursoTermInstances.hs
```

The instances of class `Term` *and* `Typeable` *are used only internally by the strategy combinators of Strafunski. As a user of these combinators, you do not need to understand them.*

Using Strafunski, the same query can be implemented more concisely as follows:

```
...
import StrategyLib
import CursoTermInstances
...

studentNames'
  = processQ (applyTU q) "teste.xml"
    where
      q = stop_tdTU (monoTU qAluno)
      qAluno (Aluno _ nome) = return [nome]
```

The traversal strategy `stop_tdTU` is used to search for students in a top down fashion, without continuing the search on the subelements of a student. The `monoTU` combinator turns the monomorphic function `qAluno` into a generic function that succeeds on students, but fails on other elements.

Exercise 4.5 *Verify that the reformulated query produces the same results as the original one. Make sure you understand the reformulated query.*

Apart from querying the document, we can transform it. For instance, we can update the attribute `n` of the element `Inscritos` such that we are sure that it matches the actual number of students registred for the course.

Exercise 4.6 *Complete the following code. And run it on the document* `teste.xml`*.*

The following code traverses the document to find elements of type `Inscritos`. These elements are subsequently queried to find the number of students that it holds. Finally, the attribute of the element `Inscritos` is updated to the computed number of students.

```
countAlunos
 = applyTU ( ... )
   where
     countAluno (a::Aluno) = return (1::Int)

updateInscritosN
  = applyTP( ... )
    where
      update (Inscritos attrs alunos)
        = do n <- countAlunos alunos
             let attrs' = attrsinscritosN=Just (show n)
             return (Inscritos attrs' alunos)
```

Note the use of record update to perform attribute updating.