MicroJava

Automatic ExtractClass refactoring using Stratego and Spoofax

Thomas Schmorleiz@SLT Koblenz

Agenda

- Extremely compact and incomplete introduction to Stratego and Spoofax using simple application for MicroJava

- ExtractClass Refactoring
  - Identification of necessary concrete transformation steps
  - Term annotation
  - Actual transformation

Stratego / Spoofax

Stratego

“Stratego/XT is a language and toolset for program transformation. The Stratego language provides rewrite rules for expressing basic transformations, programmable rewriting strategies for controlling the application of rules […].”

[http://strategoxt.org/Stratego/WebHome]

Spoofax

"With the Spoofax/IMP language workbench, you can write the grammar of your language using the high-level SDF grammar formalism. [...] services such as error marking and content completion can be specified using rewrite rules in the Stratego language."

[http://strategoxt.org/Spoofax/WebHome]

Language definition using SDF

SDF

```
context-free start-symbols
    Start

  context-free syntax

    %% programs
      "module" UCID "{" Class* "}"   -> Start {cons("Module")}

    %% classes
    "class" ID "{" Member* "}"        -> Class {cons("Class")}

    %% fields
    NTBinding                         -> Member {cons("Field")}

    %% expressions
    ...
        INT                           -> Expr {cons("NatConst")}
    Expr "+" Expr                     -> Expr {cons("Add"), assoc}
    Expr "-" Expr                     -> Expr {cons("Sub"), left}
    Expr "*" Expr                     -> Expr {cons("Mul"), assoc}
    ...
```

ENBF-like productions

Constructor annotations for AST

special annotations to avoid ambiguity

SDF

By defining a grammar+, Spoofax gives you:

- Highlighting of operators and keywords
- Code Folding

Term Transformation using Rules

Rules

```
L : p1 -> p2
```

Where:

- L is the name of the rule
- p1 is the term to be matched
- p2 is the term created

Rules can fail or succeed. In the latter case they transform the term as specified.

Rules

```
L : p1 -> p2
with
  ...
```

Using `with` one can declare rule scope variables

Rules

```
L : p1 -> p2
where
   ...
```

Using `where` one can (additionally to the matching of the term) define strategies that have to hold for the rule be applicable

Rules

```
L(s1,...,sn|t1,...,tn) : p1 -> p2
```

Rules can be parameterized with strategies and terms

Strategies

```
L(s1,...,sn|t1,...,tn) : p1 -> p2
```

- Rules are (atomic) strategies
- Combinators can be used to define more complex strategies:
  - `s1; s2` sequence
  - `try(s)` try (never fails)
  - `top-down(s) and bottom-up(s)`
  - `repeat(s)` repeat until fail

Libraries

Various expressive libraries for:

- Lists, pairs
- Parsing
- Traversal
- I/O
- ...

Spoofax: Various interaction with Eclipse, generation of editors

Sample transformation

```
// rules for class renaming
rename-class:
  (newname, selected-name, pos, ast, _, _) ->
  ([(ast, new-ast)], [], [], [])
 with
   new-ast  :=
       <topdown(try(rename-classstep(|selected-name, newname)))> ast;

rename-classstep(|old-name, new-name):
  ClassType(old-name) -> ClassType(new-name)

rename-classstep(|old-name, new-name):
  Class(old-name,fs) -> Class(new-name, fs)
```

ExtractClass refactoring

Identification of necessary concrete transformation steps

Steps

Given: List of Fields and Methods that should be extracted to a new class. Format: Abstract Syntax

To do:

- 1. Create new class with fields and methods
- 2. Create reference from old to new class
- 3. Create delegation methods in old class
- 4. Fix references to fields and methods
- 5. Create back-link from new to old class

Steps 1,2,3 and 5

```
extract-class-fields:
  (new-cn, selected-ms, pos, ast@Module(mn,csraw), path, _) -> ([(ast, Module(mn,cs'''))], [], [], [])
  with
     cs := <topdown(try(annotate))> csraw;
    // lookup target class
    parentClassIndex := <index> (2, pos);
     Class(cn, old-ms) := <index> (<inc> parentClassIndex, cs);
     // new members
     diff-ms := <diff(\(a,b) -> <eq> (<topdown(try(rm-annotations))> a, b)\)> (old-ms, selected-ms);
     delegators := <map(\Method(mn,ps,t,_) ->
                         Method(mn,ps,t,
                              [Return(Call(RefCascade(["this", <lower-case> new-cn]),mn,<map(unpack-nt-n)> ps))])\)>
              <filter(?Method(_,_,_,_))> selected-ms;
     ms' := <concat> [[Field(NT(<lower-case> new-cn, ClassType(new-cn)))], diff-ms, delegators];
     // create new class
     new-class := Class(new-cn, <concat>
              [[Field(NT(<lower-case> cn, ClassType(cn)))], <topdown(try(replace-this(|cn, ms')))> selected-ms]);
    // add new class with target members and backlink to old
    cs'   := <concat> [cs, [<topdown(try(annotate))> new-class]];
    // replace fields by class reference
    cs''  := <at-index(\x -> Class(cn, ms')\)> (parentClassIndex, cs');
```

Step 4:
Fix references to fields and methods

- References are used where the RefCascade constructor is used.

- Therefore, in when facing a reference cascade we have to know the type of all elements to decide where to replace.

- Stratego allows one to annotate terms. We annotate RefCascade with a list of types

Step 4:
Fix references to fields and methods

- What will do is to go top-down over the AST and try to apply annotation where possible.

- We have to “collect” type information. But:

  ```
  Method(mn,ps,t,ss) -> Method(mn,ps,t,newss)
  ```

  How to get the types of fields?

- Solution: Stratego allows one to define rules within the definition of other rules dynamically

Step 4:
Fix references to fields and methods

```
annotate:
   Class(cn,ms) -> Class(cn,ms)
   with
         fs := <map(\Field(x) -> x\)> <filter(?Field(_))> ms;
         rules (
              annotate:
                Method(mn,ps,t,ss) -> Method(mn,ps,t,newss)
                with
                    nts-assoc := <concat> [<map(unpack-nt)> ps, <map(unpack-nt)> fs,[("this",ClassType(cn))]];
                    newss := <Fst> <foldr(!([],nts-assoc), (add-nt <+ ann-step))> <reverse> ss

              add-nt:
           (s@Declare(n,t),(ss,nts)) -> (<concat> [ss,[s]], <concat> [[(n,t)],nts])

              ann-step:
           (s,(ss,nts)) -> (<concat> [ss,[news]], nts)
           with
              news := <topdown(try(annote-cas(|nts)))> s

            annote-cas(|nts):
              x@RefCascade([ref|refs]) ->  RefCascade([ref|refs]) {<concat> [[st], annos]}
                where
                         <lookup> (ref,nts) => st;
                         fold := <foldr(!([],st), cas-step)> <reverse> refs;
                         annos := <Fst> fold

            cas-step:
                (ref, (annos,ClassType(t))) -> (<concat> [annos, [ct']], ct')
                where
                         <lookup> (ref, <index> (1, <bagof-GetFields> t)) => ct'
         )
```

dynamic rule

Step 4:
Fix references to fields and methods

```
 cs''' := <topdown(repeat(replace-cas-call(|new-cn, selected-ms, cn)))> cs''


replace-cas-call(|new-cn, selected-ms, cn):
   RefCascade(refs@[h|t]) {types} -> RefCascade(refs') {types'}
   where
     selected-fs := <get-field-names> selected-ms;
     accesses := <zip> (<init> types, t);
     <getfirst(\(c,field) -> <and(<eq> (c, ClassType(cn)), <member> (field, selected-fs))>\)> accesses => candidate;
     cindex := <get-index> (candidate, accesses);
     refs' := <insert> (cindex, <lower-case> new-cn, refs);
     types' := <insert> (cindex, ClassType(new-cn), types)

replace-cas-call(|new-cn, selected-ms, cn):
   Call(RefCascade(refs) {types}, mn, es) -> Call(RefCascade(refs') {types'}, mn, es)
   where
     selected-meths := <get-method-names> selected-ms;
     <eq> (ClassType(cn), <last> types);
     <member> (mn, selected-meths);
     refs' := <insert> (<length> refs, <lower-case> new-cn, refs);
     types' := <insert> (<length> types, ClassType(new-cn), types)
```

Questions, Feedback so far?