Software Productivity

Paulo Borba
Informatics Center
Federal University of Pernambuco

phmb@cin.ufpe.br ◈ twitter.com/pauloborba

Strategy operators, lists of terms, traversals

Paulo Borba
Informatics Center
Federal University of Pernambuco

phmb@cin.ufpe.br ◈ twitter.com/pauloborba

Translating single property

```
rules
  property-to-gettersetter:
    |[ x_prop : srt ]| ->
    |[ private t x_prop;
       public t get#x_prop() {return x_prop;}
       public void set#x_prop(t x_prop) {
         this.x_prop = x_prop;
       }
    ]|
  where <builtin-java-type> srt => t
```

Translating list of properties

```
entity-to-class :
  |[ entity x_Class { prop* } ]| ->
  |[ @Entity public class x_Class {
       public x_Class () { } ...   ]|
...
```

Strategies for dealing with (property) lists

- `map(s)`, succeeds if `s` succeeds for each element of the list, yields the modified list of terms
- `filter(s)`, applies `s` to each element of a list, yielding a list with the elements resulting from the successful application of `s`
- `concat`, turns a list of lists into a list of elements

Composing strategies that manipulate lists of terms

```
strategies
  generate-gets-and-sets =
    map(property-to-gettersetter) ; concat

  mapconcat(s) = map(s) ; concat

  generate-gets-and-sets-equivalent =
    mapconcat(property-to-gettersetter)
```

Translating entities

```
entity-to-class :
  |[ entity x_Class { prop* } ]| ->
  |[ @Entity public class x_Class {
        public x_Class () { }
        @Id @GeneratedValue private Long id;
        public Long getId() { return id; }
        private void setId(Long id) {
         this.id = id;
        }
        ~*cbds
     }
   ]|
where <generate-gets-and-sets> prop* => cbds
```

Concrete and abstract syntax in rules

```
property-to-gettersetterConcrete:
  |[ x_prop : srt ]| -> ...
property-to-gettersetterAbstract:
  Property(x_prop,srt) -> ...
```

```
addBlockAbstract:
  If(e, stm) -> If(e, Block([stm]))
    where <not(?Block(_))> stm
addBlockConcreteAbstract:
  If(e, stm) -> |[ if (e) {stm} ]|
    where <not(?Block(_))> stm
```

Basic strategies

- `id`, returns the term to which it is applied
- `fail`, always fails

- `?(t)`, fails if the term to which it is applied does not match `t`, otherwise returns the term and binds variables
- `!(t)`, yields the term `t`

Basic strategy operators

- `s ; t`, strategy (function) composition, applies `s` to the term to which it is applied and the result is applied to `t`, fails when `s` or `t` fails
- `s < r + t`, guarded left choice operator, if `s` succeeds `r` is applied to its result, else `t` is applied to the original term, fails when `r` or `t` is selected and fails
- `s + t`, choice

Many other operators are a combination...

```
not(s)          = s < fail + id
s1 <+ s2        = s1 < id + s2
try(s)          = s <+ id
repeat(s)       = try(s; repeat(s))
<s> p           = !p ; s
s => p          =  s ; ?p
<s> p1 => p2 = !p1 ; s; ?p2
```

Including rules!

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

is syntactic sugar for

```
L = ?p1; where(s); !p2
```

term variable scope

```
where(s)
```

is syntactic sugar for

```
{x: ?x; s; !x}
```

Reflective operators

- **`all(s),`** applies **`s`** to each direct subterm of a given constructor, fails if the application to one of the subterms fails

- **`one(s),`** applies **`s`** to exactly one direct subterm

- **`some(s),`** applies **`s`** to as many direct subterms as possible and at least one, fails if the application to all of the subterms fails

Traversals are a combination...

```
topdown(s)          = s; all(topdown(s))
bottomup(s)         = all(bottomup(s)); s
innermost(s)        = bottomup(try(
                        s; innermost(s)))
alltd(s)            = s <+ all(alltd(s))
downup(s)           = s; all(downup(s)); s
downup(s1, s2)      = s1;
                      all(downup(s1, s2));
                      s2
```

exhaustive

Preserving terms

```
if s1 then s2 else s3 end
    =  where(s1) < s2 + s3

or(s1, s2) =

    if s1 then try(where(s2))
    else where(s2) end

and(s1, s2) =

    if s1 then where(s2)
    else where(s2); fail end
```

Full strategy language

```
repeat(s, c) =
   (s; repeat(s, c)) <+ c

repeat-until(s, c) =
   s; if c then id
      else repeat-until(s, c) end

while(c, s) =
   if c then s; while(c, s) end

do-while(s, c) =
   s; if c then do-while(s, c) end
```

Switch

```
switch s0
  case s1 : s1'
  case s2 : s2'
     ...
  otherwise : sdef
end
```

Strategy operators, lists of terms, traversals

Paulo Borba
Informatics Center
Federal University of Pernambuco

phmb@cin.ufpe.br ◈ twitter.com/pauloborba

Software Productivity

Paulo Borba
Informatics Center
Federal University of Pernambuco

phmb@cin.ufpe.br ◈ twitter.com/pauloborba