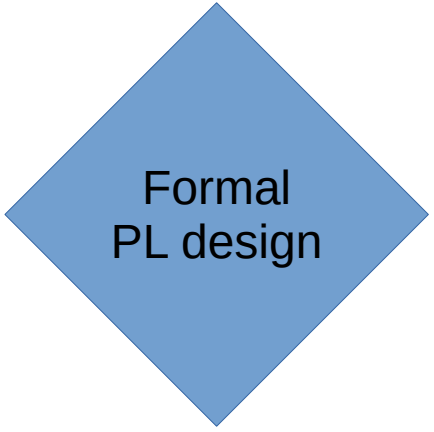
SWEN423 - Featherweight Java



Lect3: Reduction

Engagement 1 results

- 5 students not in wellington,
- 12 students chose to either not attend or have low attendance
- 10 just did not submit
  - I still have not heard back from many of them... :-(
- 5 submitted but missed the surprise.

Engagement 1 results

- Added a direct link to “Separating Use and Reuse to Improve Both”
- Zoom url for remote office hours: https://vuw.zoom.us/my/servetto
- Clarification: all submissions must contain the 'surprise of the week'
- Clarification: we do **not** have the 3 late days. We only have 1 assignment. The other submissions are Term Tests and need to be completed promptly.
- Questions “how different programming language features are well-suited to different project requirements”
- Course material: is it “something of value that I can use in the workforce”?
- Clarification: this year, as for university requirement, in person attendance is not and can not be mandatory
- Office hours as **on the slides**. There must be some very old Marco website with old office hours somewhere..

Double delivery

- Given the situation of this year
  I will do double delivery.
- Until further notice,
  Lectures will continue in class in person
- Since the recording quality is quite bad, I will re-record the lectures from Lect3 (today) with higher quality, so if you want to review the course material, or you can not be present in the class, you can rely on this content.

FJ Grammar

e ::= x | e_0.m(e_1..e_n) | new C(e_1..e_n) | e.f

cd ::= class C implements C_1..C_n { Fs K Ms }
| interface C extends C_1..C_n { MH_1; .. MH_k; }

Fs ::= F_1..F_n

Ms ::= M_1..M_n

K ::= C(C_1 x_1 .. C_n x_n){this.x_1=x_1; .. this.x_n=x_n;}

F ::= C f;

M ::= MH{return e;}

MH ::= C m(C_1 x_1 .. C_n x_n)

v ::= new C(v_1..v_n) 

Examples

```
class C { D d(){return new D(this);} }
class D { C foo; }

new C().d().foo  --> ??????????????
```

```
e  ::= x | e.m(es) | new C(e) | e.f

cd ::= class C implements Cs{ Fs K Ms }
     | interface C extends Cs{ MH1; .. MHk; }

K  ::= C(C1 x1 .. Cn xn){this.x1=x1;..this.xn=xn;}

F  ::= C f;        M  ::= MH{return e;}

MH ::= C m(C1 x1 .. Cn xn)

v  ::= new C(vs)
```

```
e  ::= x | e.m(es) | new C(e) | e.f

cd ::= class C implements Cs{ Fs K Ms }
     | interface C extends Cs{ MH1; .. MHk; }

K  ::= C(C1 x1 .. Cn xn){this.x1=x1;..this.xn=xn;}

F  ::= C f;        M  ::= MH{return e;}

MH ::= C m(C1 x1 .. Cn xn)

v  ::= new C(vs)
```

Examples

```
class C { D d(){return new D(this);} }
class D { C foo; }
```

`new C().d()`.foo --> `????????????` .foo

Redex

Redex: the smallest sub-expression that can be reduced

```
e  ::= x | e.m(es) | new C(e) | e.f

cd ::= class C implements Cs{ Fs K Ms }
     | interface C extends Cs{ MH1; .. MHk; }

K  ::= C(C1 x1 .. Cn xn){this.x1=x1;..this.xn=xn;}

F  ::= C f;        M  ::= MH{return e;}

MH ::= C m(C1 x1 .. Cn xn)

v  ::= new C(vs)
```

Examples

```
class C { D d(){return new D(this);} }
class D { C foo; }
```

new C().d().foo --> ???????????? .foo

Value

Value:
a fully
reduced expression

```
e  ::= x | e.m(es) | new C(e) | e.f

cd ::= class C implements Cs{ Fs K Ms }
     | interface C extends Cs{ MH1; .. MHk; }

K  ::= C(C1 x1 .. Cn xn){this.x1=x1;..this.xn=xn;}

F  ::= C f;        M  ::= MH{return e;}

MH ::= C m(C1 x1 .. Cn xn)

v  ::= new C(vs)
```

Examples

```
class C { D d(){return new D(this);} }
class D { C foo; }
```

`new C().d().foo` --> `new D(new C()).foo`

Redex

Value

Redex: the smallest sub-expression that can be reduced

```
e  ::= x | e.m(es) | new C(e) | e.f

cd ::= class C implements Cs{ Fs K Ms }
     | interface C extends Cs{ MH1; .. MHk; }

K  ::= C(C1 x1 .. Cn xn){this.x1=x1;..this.xn=xn;}

F  ::= C f;        M  ::= MH{return e;}

MH ::= C m(C1 x1 .. Cn xn)

v  ::= new C(vs)
```

Examples

```
class C { D d(){return new D(this);} }
class D { C foo; }
```

`new C().d().foo --> new D(new C()).foo --> new C()`

Redex

Value

Redex: the smallest sub-expression that can be reduced

```
e  ::= x | e.m(es) | new C(e) | e.f

cd ::= class C implements Cs{ Fs K Ms }
     | interface C extends Cs{ MH1; .. MHk; }

K  ::= C(C1 x1 .. Cn xn){this.x1=x1;..this.xn=xn;}

F  ::= C f;        M  ::= MH{return e;}

MH ::= C m(C1 x1 .. Cn xn)

v  ::= new C(vs)
```

Examples

```
class C { D d(){return new D(this);} }
class D { C foo; }
```

`new C().d()`.foo --> `new D(new C())`.foo --> new C()

Redex

Value

Redex: the smallest sub-expression that can be reduced

Grammar of the reduction:

e_1 --> e_2

The reduction relation is the set of all valid terms of form e_1 --> e_2

Using induction to formally define sets of syntactic elements: METARULES

```
e  ::= x | e.m(es) | new C(e) | e.f

cd ::= class C implements Cs{ Fs K Ms }
     | interface C extends Cs{ MH1; .. MHk; }

K  ::= C(C1 x1 .. Cn xn){this.x1=x1;..this.xn=xn;}

F  ::= C f;        M  ::= MH{return e;}

MH ::= C m(C1 x1 .. Cn xn)

v  ::= new C(vs)
```

A (Inductive) Rule is a way to build a set: it has (possibly empty) set of premises and a consequence. For example

```
      new C().d() -->  new D(new C())
  ----------------------------------------
    new C().d().foo --> new D(new C()).foo


                                  //empty on purpose: no premises
----------------------------------------
    new C().d() --> new D(new C())
```

.

A Rule is fully “concrete”. We would need and infinite amount of rules to encode an infinite set.

A Metarule is a rule with variables, that can also be used in some boolean side conditions.
We call such variables metavariables.

```
e  ::= x | e.m(es) | new C(e) | e.f

cd ::= class C implements Cs{ Fs K Ms }
     | interface C extends Cs{ MH1; .. MHk; }

K  ::= C(C1 x1 .. Cn xn){this.x1=x1;..this.xn=xn;}

F  ::= C f;        M  ::= MH{return e;}

MH ::= C m(C1 x1 .. Cn xn)

v  ::= new C(vs)
```

For example, this rule

$$\frac{\texttt{new C().d()} \rightarrow \texttt{new D(new C())}}{\texttt{new C().d().foo} \rightarrow \texttt{new D(new C()).foo}}$$

Can be obtained from the following Metarule by replacing 'e_1', 'e_2' and 'f'

$$(\text{f-in})\ \frac{e_1 \rightarrow e_2}{e_1.f \rightarrow e_2.f}$$

f-in is a **metarule**, e_1, e_2 and f are **metavariables**

.

FJ metarules with no premises:

(f-access)----------

$$\text{new } C(v_1..v_n).x_i \text{ --> } v_i$$

where:
  class C _{ C_1 x_1;..C_n x_n; K Ms } in cds

(m-call)----------

$$\text{new } C(vs).m(v_1..v_n) \text{ --> } e'$$

where:
  e' = e[this=new C(vs)][x_1=v_1]..[x_n=v_n]
  class C _{ _ C_0 m(C_1 x_1..C_n x_n){return e;} _ } in cds

#Define $e_0[x=e_1] = e_2$
  x[x=e] = e
  x[x'=e] = x *where:* x != x'
  e_0.m(e_1..e_n)[x=e] = e_0[x=e].m(e_1[x=e]..e_n[x=e])
  new C(e_1..e_n)[x=e] = new C(e_1[x=e]..e_n[x=e])
  e_0.f[x=e] = e_0[x=e].f

e ::= x | e.m(es) | new C(e) | e.f

cd ::= class C implements Cs{ Fs K Ms }
    | interface C extends Cs{ MH_1; .. MH_k; }

K ::= C(C_1 x_1 .. C_n x_n){this.x_1=x_1;..this.x_n=x_n;}

F ::= C f; M ::= MH{return e;}

MH ::= C m(C_1 x_1 .. C_n x_n)

v ::= new C(vs)

-underscore matches any grammar

-where for side conditions

-notations with define

-cds in the global scope

FJ metarules with premises:

$$\frac{e_1 \rightarrow e_2}{e_1.f \rightarrow e_2.f}\ (\text{f-in})$$

$$\frac{e_1 \rightarrow e_2}{e_1.m(es) \rightarrow e_2.m(es)}\ (\text{m-call-r})$$

$$\frac{e_1 \rightarrow e_2}{v.m(vs,e_1,es) \rightarrow v.m(vs,e_2,es)}\ (\text{m-call-p})$$

$$\frac{e_1 \rightarrow e_2}{new\ C(vs,e_1,es) \rightarrow new\ C(vs,e_2,es)}\ (\text{new-p})$$

.

```
e  ::= x | e.m(es) | new C(e) | e.f

cd ::= class C implements Cs{ Fs K Ms }
     | interface C extends Cs{ MH1; .. MHk; }

K  ::= C(C1 x1 .. Cn xn){this.x1=x1;..this.xn=xn;}

F  ::= C f;        M  ::= MH{return e;}

MH ::= C m(C1 x1 .. Cn xn)

v  ::= new C(vs)
```

COMPACT:
-6 metarules and 1 notation

-next, we will see how to remove 3 rules

-hopefully, all FJ could fit on a single slide

FJ Grammar, Ɛv

$$e ::= x \mid e_0.m(es) \mid \text{new } C(es) \mid e.f$$

$$cd ::= \text{class } C \text{ implements } Cs\{ Fs\ K\ Ms \} \mid \text{interface } C \text{ extends } Cs\{ MH_1; \ldots MH_k; \}$$

$$K ::= C(C_1\ x_1 \ldots C_n\ x_n)\{\text{this}.x_1=x_1; \ldots \text{this}.x_n=x_n;\}$$

$$F ::= C\ f;$$

$$M ::= MH\{\text{return } e;\}$$

$$MH ::= C\ m(C_1\ x_1 \ldots C_n\ x_n)$$

$$v ::= \text{new } C(vs)$$

$$Ɛv ::= \square \mid Ɛv.m(es) \mid v.m(vs,Ɛv,es) \mid \text{new } C(vs,Ɛv,es) \mid Ɛv.f$$

FJ Grammar, Ɛv

Ɛv ::= □ | Ɛv.m(es) | v.m(vs,Ɛv,es)| new C(vs,Ɛv,es) | Ɛv.f

“Ɛv” is the evaluation context; the v stands for “value”.

It is just “grammar”. The symbol “□” is called the hole.

A term that is syntactically a “Ɛv” will have exactly one hole. The hole is in a point that respects the designed evaluation order: every sub term on the left is a value.

#Define $Ɛv[e_1] = e_2$

□[e] = e
Ɛv.m(es)[e] = Ɛv[e].m(es)
v.m(vs,Ɛv,es)[e] = v.m(vs,Ɛv[e],es)
new C(vs,Ɛv,es)[e] = new C(vs,Ɛv[e],es)
Ɛv.f[e] = Ɛv[e].f

FJ metarules with premises using Ɛv:

$$\frac{e_1 \rightarrow e_2}{\mathcal{E}v[e_1] \rightarrow \mathcal{E}v[e_2]}\ (ctx)$$

The same without using Ɛv:

$$\frac{e_1 \rightarrow e_2}{e_1.f \rightarrow e_2.f}\ (f\text{-}in)$$

$$\frac{e_1 \rightarrow e_2}{e_1.m(es) \rightarrow e_2.m(es)}\ (m\text{-}call\text{-}r)$$

$$\frac{e_1 \rightarrow e_2}{v.m(vs,e_1,es) \rightarrow v.m(vs,e_2,es)}\ (m\text{-}call\text{-}p)$$

$$\frac{e_1 \rightarrow e_2}{new\ C(vs,e_1,es) \rightarrow new\ C(vs,e_2,es)}\ (new\text{-}p)$$

```
e  ::= x | e.m(es) | new C(e) | e.f

cd ::= class C implements Cs{ Fs K Ms }
     | interface C extends Cs{ MH1; .. MHk; }

K  ::= C(C1 x1 .. Cn xn){this.x1=x1;..this.xn=xn;}

F  ::= C f;        M  ::= MH{return e;}

MH ::= C m(C1 x1 .. Cn xn)

v  ::= new C(vs)
```

1 rule instead of 4

For larger languages, contexts are a fundamental metarules programming pattern

More kinds of contexts

(Deep) Evaluation context:

Ɛv ::= □ | Ɛv.m(es) | v.m(vs,Ɛv,es)| new C(vs,Ɛv,es) | Ɛv.f

(Deep) Full context:

Ɛ ::= □ | Ɛ.m(es) | e.m(es,Ɛ,es')| new C(es,Ɛ,es') | Ɛ.f

Shallow Evaluation context:

Ɛv ::= □.m(es) | v.m(vs,□,es)| new C(vs,□,es) | □.f

Shallow Full context:

Ɛ ::= □.m(es) | e.m(es,□,es')| new C(es,□,es') | □.f

Metarules Programming patterns

- As for any programming language, the language of metarules have useful patterns. Evaluation contexts are just a programming pattern.
- Useful when selecting sub-terms. Example:

```
_______
#Define e inside e'
  e inside Ɛ[e]    //where Ɛ is the Deep Full context
```

Metarules Programming patterns

- Example: use the Deep Full context to select an arbitrary subexpression e' of a field access contained in e:

```
e = Ɛ[Ɛ'[e'].f]
```