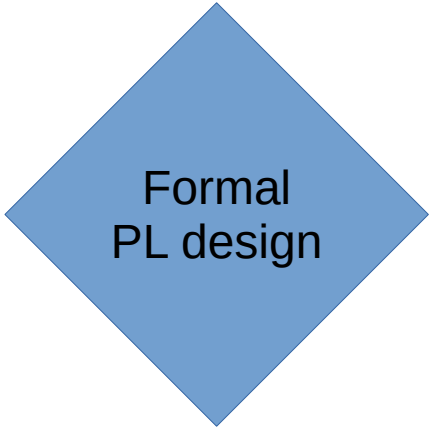
SWEN423 - Featherweight Java



Lect5: Overview FJ and Metarules

A language of pure syntax

- Metarules allows to define a syntax and then to work on syntax to define.... more syntax.
- That all there is, it is inductive symbolic manipulation of symbols.
- No attempt and no need to go above the purely syntactical level.
- Metarules are an extraordinarily compact metalanguage

Grammars are just metarules

$e ::= x \mid e_0.m(e_1..e_n) \mid \text{new } C(e_1..e_n) \mid e.f$

e

$$\frac{}{x}\,(x)$$

$$\frac{}{e_0.m(e_1..e_n)}\,(meth)$$

$$\frac{}{\text{new } C(e_1..e_n)}\,(new)$$

$$\frac{}{e_0.f}\,(f)$$

When unused, we do not specify the name of the defined set.

When needed, we can put it in a a box on top, like here

Notations are just metarules

#Define $e_0[x=e_1] = e_2$

$x[x=e] = e$

$x[x'=e] = x$ *where:* $x \mathrel{!}= x'$

$e_0.m(e_1..e_n)[x=e] = e_0[x=e].m(e_1[x=e]..e_n[x=e])$

$\text{new } C(e_1..e_n)[x=e] = \text{new } C(e_1[x=e]..e_n[x=e])$

$e_0.f[x=e] = e_0[x=e].f$

$$\frac{}{x[x=e] = e} \quad (xx)$$

$$\frac{x \mathrel{!}= x'}{x[x'=e] = x} \quad (xx')$$

$$e_0.m(e_1..e_n)[x=e] = e_0[x=e].m(e_1[x=e]..e_n[x=e]) \quad (m[])$$

$$\text{new } C(e_1..e_n)[x=e] = \text{new } C(e_1[x=e]..e_n[x=e]) \quad (n[])$$

$$e_0.f[x=e] = e_0[x=e].f \quad (f[])$$

Here is quite subtle:
We are adding elements to
the ' = ' relation.

Moreover, when the element
'a = b' is in the equality relation,
we can write the expression b
inside of the premises/side conditions
instead of the term a. For example,
we are writing $e_n[x=e]$ directly in the
metarule instead of defining extra
side conditions

Set definitions

- Many times we need to define sets, and we do it by adding elements to the 'set containment' relation “a in b”

```
_______
#Define pairs(cats)
  <cat₁,cat₂> in pairs(_,cat₁,_,cat₂,_)
    age(cat₁) ≤ age(cat₂)
  <cat₂,cat₁> in pairs(_,cat₁,_,cat₂,_)
    age(cat₂) ≤ age(cat₁)



G::= N₁(Ns₁)..Nₙ(Nsₙ)  //a grammar for a graph

_______
#Define reachable(G,N)=Ns
  N in reachable(G,N)              //N is reachable from N
  N₀ in reachable(_ N(_ N₀ _) _,N) //any element of Ns is reachable from N
  N₁ in reachable(G,N)            //N₁ is reachable
    N₀ in reachable(G,N)          //if we can reach N₀ from N
    N₁ in reachable(G,N₀)         //and N₁ from N₀
```

Set definitions

Craft::= Rs_1->R_1 .. Rs_n->R_n //2 sticks and 3 stones for a pickaxe

Map::= L_1+R_1->L'_1..L_n+R_n->L'_n //need a resource to reach a location

S::=L(Rs) //player state

#Define reachable(Map,Craft,S)=Ss canCraft(Craft,Rs)=Rs'

S in reachable(_,_,S)

L(Rs') in reachable(_, Craft,L(Rs))

Rs' in canCraft(Craft,Rs)

L'(Rs) in reachable(_ L+R->L_0 _, Craft,L(Rs))

R in Rs

S_2 in reachable(Map,Craft,Rs,S_0)

S_1 in reachable(Map,Craft,Rs,S_0)

S_2 in reachable(Map,Craft,Rs,S_1)

Rs'\{Rs} R in canCraft(_ Rs->R _, Rs')

{Rs} contained in Rs'

Proving properties about programming languages

- The metarule language is not just very compact; it is also based in induction, thus it is feasible to use it to prove properties of programming languages. In 2020, the metarule language is the de-facto standard to prove properties of programming languages; manual proofs are doing directly on it, while automated proofs (for examples using Coq and Isabelle) relies on encoding it.

Proving properties about programming languages

- The most common property we are interested in a language is “Soundness”.
- Informally, soundness means “we considered all the cases”; the specification of our language completely specify the behavior of the language in all the cases.

Values, Redexes and Stuck

- Given a reduction arrow, a **normal form** is a term that can not be reduced any further.
- A **value** is a kind of normal form identified as an expected final result.
- Any normal form that is not a value is called **Stuck.**
- For example, calling a method that does not exists.
- The goal of **type systems** is to rule-out terms whose execution may go stuck.

Soundness for a typed language

- Given a class table `cds` and an expression `e`, where `∅ |- e : C` and forall `cd` in `cds cd : ok` if `e -->*e'` and not `e' -->_` then `e'` is of form `v`.

- **In natural language:** if the class table is well typed and the main expression is well typed without free variables, then either the execution loops forever or it reduces to a value

  //Note: `∅  is the empty set`

Soundness = Progress + Preservation

- One of the most common ways to prove soundness is to divide it in two properties:
- Progress: “can do a step”

- Preservation: “doing a step preserves typing”

Progress

- Given a class table `cds` and an expression `e`, where `∅ |- e : C` and forall `cd` in `cds cd : ok` either `e'` is of form `v or e --> e'`

- **In natural language:** if the class table is well typed and the main expression is well typed without free variables, then either the expression is already a value, or can be reduced one step.

Preservation (Also called subject reduction)

- Given a class table `cds` and an expression `e`, where `∅ |- e : C` and forall `cd` in `cds cd : ok` if `e --> e' then ∅ |- e' : C`

- **In natural language:** if the class table is well typed and the main expression is well typed without free variables, if the expression can be reduced one step, then the reduced expression is still well typed.

What is next?

- Adding more conventional features to FJ:
  - Exceptions, Generics, field update, local variables, local variable update, inheritance, interface default methods, lambdas
- Adding novel features to FJ:
  - reference capabilities (mut, imm, read, lent, capsule), strong exception safety, multimethods, trait composition, circular initialization support, automatic parallelism, ...
- Making proofs of soundness of those features!