Concrete Redex: semantics engineering with concrete syntax

Tijs van der Storm
storm@cwi.nl / @tvdstorm

CWI
Centrum Wiskunde & Informatica

university of groningen

Semantics Engineering

- Defining, simulating, debugging, evaluating semantics of programming languages

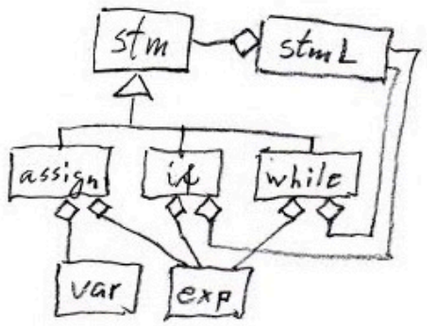
- Goal: semantics engineering as integrated part of language workbench offering

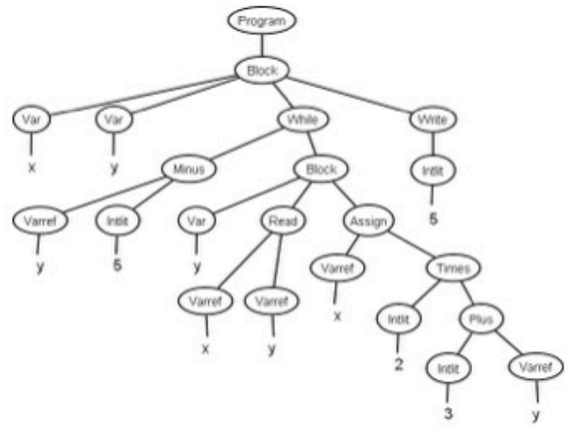
- Rascal: http://www.rascal-mpl.org :)

Concrete Redex

- Like Redex (https://redex.racket-lang.org/)
- Evaluation Context Semantics (small step)
- Use parsing for context decomposition (!)
- Slow as hell, but really cool ;)

Why?

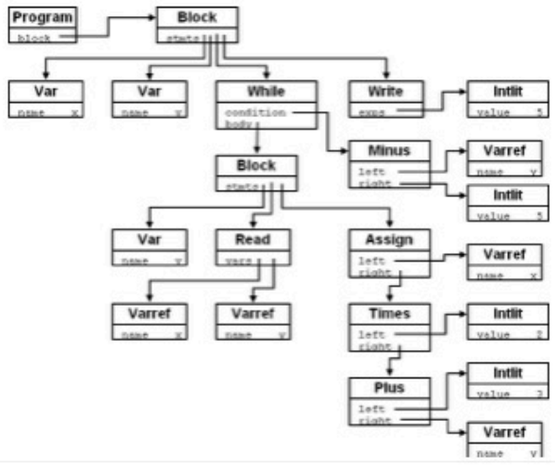


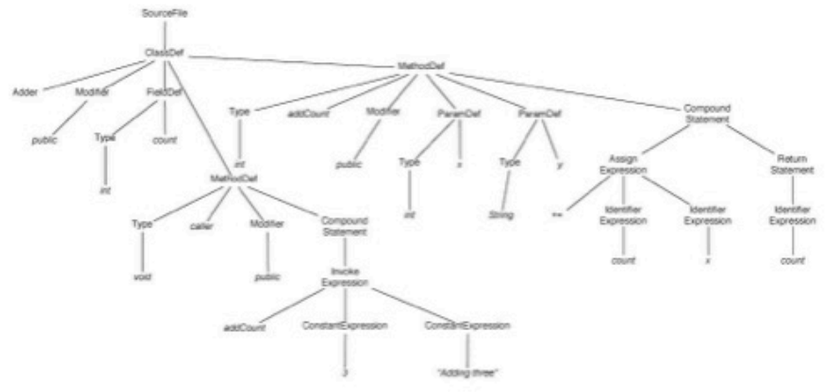


Abstract Syntax Sucks!

deconstruction (?), allegory (?), ...

Tijs van der Storm





WGLD2.16
Aarhus, 2013

```
(y := y + [1] + y;
if x <= y then
  x := x + y
else
  y := 0
fi, [x ↦ 1, y ↦ 0])
```

```
(y := [0] + x + y;
if x <= y then
  x := x + y
else
  y := 0
fi, [x ↦ 1, y ↦ 0])
```

```
(y := y + x + [0];
if x <= y then
  x := x + y
else
  y := 0
fi, [x ↦ 1, y ↦ 0])
```

lookup: y

lookup: y

lookup: x

lookup: y

lookup: x

lookup: y

```
(y := [0] + [1] + y;
if x <= y then
  x := x + y
else
  y := 0
fi, [x ↦ 1, y ↦ 0])
```

```
(y := y + [1] + [0];
if x <= y then
  x := x + y
else
  y := 0
fi, [x ↦ 1, y ↦ 0])
```

```
(y := [0] + x + [0];
if x <= y then
  x := x + y
else
  y := 0
fi, [x ↦ 1, y ↦ 0])
```

add: 0 + 1

lookup: y

lookup: y

lookup: x

```
(y := [1] + y;
if x <= y then
  x := x + y
else
  y := 0
fi, [x ↦ 1, y ↦ 0])
```

```
(y := [0] + [1] + [0];
if x <= y then
  x := x + y
else
  y := 0
fi, [x ↦ 1, y ↦ 0])
```

Structural operational semantics

$$\text{[LEFT]}\quad \frac{E_0 \longrightarrow E_0'}{E_0 \times E_1 \longrightarrow E_0' \times E_1} \qquad \text{[MUL]}\quad \frac{}{z_0 \times z_1 \longrightarrow z}\ z = z_0 z_1$$

$$\text{[RIGHT]}\quad \frac{E_1 \longrightarrow E_1'}{E_0 \times E_1 \longrightarrow E_0 \times E_1'} \qquad \text{[COND]}\quad \frac{E_0 \longrightarrow E_0'}{E_0 \,?\, E_1 : E_2 \longrightarrow E_0' \,?\, E_1 : E_2}$$

$$\text{[LEFT}_0\text{]}\quad \frac{}{0 \times E_1 \longrightarrow 0} \qquad \text{[COND}_\text{Z}\text{]}\quad \frac{}{z \,?\, E_1 : E_2 \longrightarrow E_2}\ z = 0$$

$$\text{[RIGHT}_0\text{]}\quad \frac{}{E_0 \times 0 \longrightarrow 0} \qquad \text{[COND}_\text{NZ}\text{]}\quad \frac{}{z \,?\, E_1 : E_2 \longrightarrow E_1}\ z \neq 0$$

$$
\begin{array}{rcl}
Context & ::= & \square \\
& | & Context \texttt{ + } AExp \mid AExp \texttt{ + } Context \\
& | & Context \texttt{ / } AExp \mid AExp \texttt{ / } Context \\
& | & \\
& | & Context \texttt{ <= } AExp \mid Int \texttt{ <= } Cxt \\
& | & \texttt{not } Context \\
& | & Context \texttt{ and } BExp \\
& | & Id \texttt{ := } Context \\
& | & Context \texttt{ ; } Stmt \\
& | & \texttt{if } Context \texttt{ then } Stmt \texttt{ else } Stmt
\end{array}
$$

$$
\begin{array}{rcl}
AExp & ::= & Int \mid Id \mid \\
& | & AExp \texttt{ + } AExp \\
& | & AExp \texttt{ / } AExp \\
BExp & ::= & Bool \\
& | & AExp \texttt{ <= } AExp \\
& | & \texttt{not } BExp \\
& | & BExp \texttt{ and } BExp \\
Stmt & ::= & Id \texttt{ := } AExp \\
& | & Stmt \texttt{ ; } Stmt \\
& | & \texttt{if } BExp \texttt{ then } Stmt \texttt{ else } Stmt \\
& | & \texttt{while } BExp \texttt{ do } Stmt \\
Pgm & ::= & \texttt{var } \mathbf{List\{}Id\mathbf{\}} \texttt{ ; } Stmt
\end{array}
$$

$$
\begin{aligned}
\langle c, \sigma \rangle [x] &\to \langle c, \sigma \rangle [\sigma(x)] \quad \text{if } \sigma(x) \neq \bot \\
i_1 \texttt{ + } i_2 &\to i_1 +_{Int} i_2 \\
i_1 \texttt{ / } i_2 &\to i_1 /_{Int} i_2 \quad \text{if } i_2 \neq 0 \\
i_1 \texttt{ <= } i_2 &\to i_1 \leq_{Int} i_2 \\
\texttt{not true} &\to \texttt{false} \\
\texttt{not false} &\to \texttt{true} \\
\texttt{true and } b_2 &\to b_2 \\
\texttt{false and } b_2 &\to \texttt{false} \\
\langle c, \sigma \rangle [x \texttt{ := } i] &\to \langle c, \sigma[i/x] \rangle [\texttt{ skip }] \quad \text{if } \sigma(x) \neq \bot \\
\texttt{skip ; } s_2 &\to s_2 \\
\texttt{if true then } s_1 \texttt{ else } s_2 &\to s_1 \\
\texttt{if false then } s_1 \texttt{ else } s_2 &\to s_2 \\
\texttt{while } b \texttt{ do } s &\to \texttt{if } b \texttt{ then } (s \texttt{ ; while } b \texttt{ do } s) \texttt{ else skip} \\
\langle \texttt{ var } xl \texttt{ ; } s \rangle &\to \langle s, (xl \mapsto 0) \rangle
\end{aligned}
$$

Operational semantics with evaluation contexts

- Separate traversal boilerplate from reduction steps
- Traversal is defined using a context grammar
  - **split** a term into a one-whole context and redex
  - perform the reduction on the redex
  - **plug** the reduct back into the context

Everything is syntax!

- Let’s put our money where our mouth is ;)
- Context-free **grammars** for context grammars
- Generalized **parsing** for context decomposition
- Non-unique decomposition == **ambiguity**! 😜

Lambda calculus

```
syntax Expr
   = var: Id \ Reserved
   | val: Value
   | app: "(" Expr+ ")";

syntax Value
   = lam: "(" "λ" "(" Id ")" Expr ")"
   | num: Num
   | add: "+";
```

```
// Contexts
syntax E
   = "(" Value* E Expr* ")"
   | @hole "(" Value Value* ")"
;
```

enforce left-to-right evaluation

the syntax of holes, or redexes

Reduction rules

The reduction relation “red”

Label

Capturing the context E

Concrete syntax pattern for the redex

```
CR red("+", E e, (E)`(+ <Num n1> <Num n2>)`)
  = {<e, [Expr]"<toInt(n1) + toInt(n2)>">};
```

Returning original context (unmodified here)

The reduct: do the addition and parse to Expr

```
CR red("+", E e, (E)`(+ <Num n1> <Num n2>)`)
  = {<e, [Expr]"<toInt(n1) + toInt(n2)>">};

CR red("βv", E e, (E)`((λ (<Id x>) <Expr b>) <Value v>)`)
  = {<e, subst((Expr)`<Id x>`, (Expr)`<Value v>`, b)>};
```

generic substitution function (capture avoiding)

```
default CR red(str _, E _, Tree _) = {};
```

catch all case: “stuck”

Concrete redex process

- parse over context-grammar: parse forest with redexes annotated with @hole
- split each parse tree into context/redex pair
- apply all applicable rules to context/redex pairs
- plug reducts back into the context terms
- unparse all plugged terms to text
- repeat until “stuck”

Demo

More goodies

- Concrete redex definitions are modular, because Rascal's grammars and functions are extensible

- Given name analysis, get capture avoiding substitution for free

EDSL for name resolution

```
void resolve((Expr)`<Id x>`, Resolver r) = r.refer(x);

void resolve((Expr)`(λ (<Id x>) <Expr e>)`, Resolver r) {
  scope(e, () {
    r.declare(x);
    r.resolve(e);
  });
}

default void resolve(Expr e, Resolver r)
  = r.recurse(e);
```

```
// replace x with e in t
Expr subst(Expr x, Expr e, Expr t)
  = subst(#Expr, (x: e), t, resolve);
```

Open issues

- Improve performance
  - current observation: parsing/unparsing is not the bottleneck, backtracking is
- How to deal with priorities/precedence in context-grammars?

Concrete Redex

- Semantics engineering => language workbenches
- Concrete Redex: library for defining evaluation context semantics of a language in Rascal
- Key feature: use concrete syntax and parsing
- “Much more” user friendly interface for semantics engineers