

## Evaluation Strategies

©Andrey Kruglyak, 2010

Wednesday, April 21, 2010

1

## Operational Semantics of $\lambda$ -calculus

$$\frac{t_2 \rightarrow t_2'}{t_1 t_2 \rightarrow t_1 t_2'} \text{ (App1)}$$

$$\frac{}{(\lambda x. t_1) v \rightarrow [x \mapsto v] t_1} \text{ (App2)}$$

$$\frac{t_1 \rightarrow t_1'}{t_1 + t_2 \rightarrow t_1' + t_2} \text{ (Add1)}$$

$$\frac{t_2 \rightarrow t_2'}{v + t_2 \rightarrow v + t_2'} \text{ (Add2)}$$

$$\frac{v_1 + v_2 = v_3}{v_1 + v_2 \rightarrow v_3} \text{ (Add3)}$$

$$\frac{t_1 \rightarrow t_1'}{t_1 - t_2 \rightarrow t_1' - t_2} \text{ (Sub1)}$$

$$\frac{t_2 \rightarrow t_2'}{v - t_2 \rightarrow v - t_2'} \text{ (Sub2)}$$

$$\frac{v_1 - v_2 = v_3}{v_1 - v_2 \rightarrow v_3} \text{ (Sub3)}$$

Wednesday, April 21, 2010

2

## Order of evaluation

- Looking at the rules Add1-3 and Sub1-3, we can see that our operational semantics specifies evaluation of sums and differences from left to right. Alternatively, we could specify evaluation from right to left, or we could allow both. In this particular case with '+' and '-', the result would be the same.
- For function applications, the order of evaluation of the function's body and its arguments is particularly important (we can evaluate the arguments before evaluating the body of the function, or substitute them into the body without evaluation).

Wednesday, April 21, 2010

3

## Example

- `int x = 0;`
- `int addToX(int a) {  
  x += a;  
  return x;  
}`
- `int fun(int a) {  
  int k = x;  
  return k+a;  
}`
- `m = fun(addToX(3));`
- Here  $m = 6$  if the argument "addToX(3)" is evaluated before the body of the function "fun", and  $m = 3$  if the argument is evaluated first when it is used

Wednesday, April 21, 2010

4

## Evaluation strategy

- ◁ In fact, this property is so important that languages are characterized by their evaluation strategy:
- ◁ Call-by-value (CBV): arguments to a function are always evaluated before the body of the function (most languages, including our  $\lambda$ -calculus, adopt this strategy)
- ◁ Call-by-name (CBN): arguments are substituted and only evaluated when needed (may lead to multiple evaluation if an argument is used several times) (e.g. Algol-60)
  - ◁ Call-by-need: a variant of CBN where arguments are substituted but only evaluated once (e.g. Haskell)

## Operational Semantics of $\lambda$ -calculus

~~$$\frac{t_2 \rightarrow t_2'}{t_1 t_2 \rightarrow t_1 t_2'} \quad (\text{App1})$$~~

$$\frac{}{(\lambda x. t_1) t_2 \rightarrow [x \mapsto t_2] t_1} \quad (\text{App2})$$

$$\frac{t_1 \rightarrow t_1'}{t_1 + t_2 \rightarrow t_1' + t_2} \quad (\text{Sum1})$$

$$\frac{t_2 \rightarrow t_2'}{v + t_2 \rightarrow v + t_2'} \quad (\text{Sum2})$$

$$\frac{v_1 + v_2 = v_3}{v_1 + v_2 \rightarrow v_3} \quad (\text{Sum3})$$

Call-by-name

$$\frac{t_1 \rightarrow t_1'}{t_1 - t_2 \rightarrow t_1' - t_2} \quad (\text{Dif1})$$

$$\frac{t_2 \rightarrow t_2'}{v - t_2 \rightarrow v - t_2'} \quad (\text{Dif2})$$

$$\frac{v_1 - v_2 = v_3}{v_1 - v_2 \rightarrow v_3} \quad (\text{Dif3})$$

## Strict vs. lazy evaluation

- ◁ CBV evaluation is strict (as opposed to non-strict, or lazy), i.e. arguments are always evaluated whether or not they are used in the body of the function
- ◁ This is important if argument evaluation may produce side-effects, for example, if it modifies a global variable

## Other evaluation strategies

- ◁ Full  $\beta$ -reduction - any function application can be reduced at any time
- ◁ Normal order - leftmost, outermost function application is reduced first, reductions within abstractions are allowed
- ◁ Call-by-name evaluation follows normal order but no reductions inside abstractions are allowed.