F5

Semantics of expressions

Lennart Andersson

Revision 2009-03-29

2009

While language

Abstract grammar (Nielson style)

$$
\begin{array}{lll}
a & ::= & n \mid x \mid a_1 + a_2 \mid a_1 * a_2 \mid a_1 - a_2 \\
b & ::= & \mathit{true} \mid \mathit{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2 \\
S & ::= & x := a \mid \mathit{skip} \mid S_1; S_2 \mid \mathit{if}\ b\ \mathit{then}\ S_1\ \mathit{else}\ S_2 \mid \mathit{while}\ b\ \mathit{do}\ S
\end{array}
$$

n will range over numerals, **Num**

x will range over variables, **Var**

a will range over arithmetic expressions, **Aexp**

b will range over boolean expressions, **Bexp**

S will range over statements, **Stm**

Numerals

Abstract grammar for **Num**

$$n ::= 0 \mid 1 \mid n\,0 \mid n\,1$$

Semantics, $\mathcal{N} \in \mathbf{Num} \to \mathbb{N}$

$$
\begin{array}{l}
\mathcal{N}[\![0]\!] = 0 \\
\mathcal{N}[\![1]\!] = 1 \\
\mathcal{N}[\![n\,0]\!] = 2 * \mathcal{N}[\![n]\!] \\
\mathcal{N}[\![n\,1]\!] = 2 * \mathcal{N}[\![n]\!] + 1
\end{array}
$$

Inductive definitions

- Basis cases are not recursive.
- Inductive cases are recursive but the recursive use of a name represents something "closer" to basis cases.
- All cases must be disjunct.

Inductive definitions – Non-example

$$f(n) = \begin{cases} 1 & \text{if } n = 1 \\ f(n/2) & \text{if } n \text{ is even} \\ f(3 * n + 1) & \text{otherwise} \end{cases}$$

Structural induction

Given an abstract grammar and a property.

1. Prove that all basis elements have the property.
2. Prove for all inductive cases that the composite element has the property assuming that all the components have the property.

Arithmetic expressions – Grammar

$n \in$ **Num** numerals

$x \in$ **Var** variables

$a \in$ **Aexp** arithmetic expressions

$$a ::= n \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 - a_2$$

```
data Aexp = Num Int | Var String | Add Aexp Aexp |
            Mul Aexp Aexp | Sub Aexp Aexp
```


Arithmetic expressions – Semantics

$\mathbf{State} = \mathbf{Var} \to \mathbb{Z}$

$\mathcal{A} \in \mathbf{Aexp} \to \mathbf{State} \to \mathbb{Z}$

$\mathcal{A}[\![n]\!]\ \sigma = \mathcal{N}[\![n]\!]$

$\mathcal{A}[\![x]\!]\ \sigma = \sigma\ x$

$\mathcal{A}[\![a_1 + a_2]\!]\ \sigma = \mathcal{A}[\![a_1]\!]\ \sigma + \mathcal{A}[\![a_1]\!]\ \sigma$

$\mathcal{A}[\![a_1 * a_2]\!]\ \sigma = \mathcal{A}[\![a_1]\!]\ \sigma * \mathcal{A}[\![a_1]\!]\ \sigma$

$\mathcal{A}[\![a_1 - a_2]\!]\ \sigma = \mathcal{A}[\![a_1]\!]\ \sigma - \mathcal{A}[\![a_1]\!]\ \sigma$

Arithmetic expressions – Implementation

```
type State = String -> Int

valA :: Aexp -> State -> Int

valA (Num n) state = n
valA (Var x) state = state x
valA (Add a1 a2) state = valA a1 state + valA a2 state
valA (Mul a1 a2) state = valA a1 state * valA a2 state
valA (Sub a1 a2) state = valA a1 state - valA a2 state
```


Boolean – Grammar

$b \in$ **Bexp** boolean expressions

$b ::= \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2$

```
data Bexp = True' | False' | Equal Aexp Aexp |
            LessEq Aexp Aexp | Not Bexp | And Bexp Bexp
```


Boolean – Semantics

$\mathcal{B} \in$ **Bexp** $\rightarrow$ **State** $\rightarrow \mathbb{B}$

$\mathcal{B}[\![\texttt{true}]\!]\ \sigma = \mathbf{tt}$
$\mathcal{B}[\![\texttt{false}]\!]\ \sigma = \mathbf{ff}$
$\mathcal{B}[\![a_1 = a_2]\!]\ \sigma = (\mathcal{A}[\![a_1]\!]\ \sigma = \mathcal{A}[\![a_1]\!]\ \sigma)$
$\mathcal{B}[\![a_1 \leq a_2]\!]\ \sigma = (\mathcal{A}[\![a_1]\!]\ \sigma \leq \mathcal{A}[\![a_1]\!]\ \sigma)$
$\mathcal{B}[\![b_1 \wedge b_2]\!]\ \sigma = \mathcal{B}[\![b_1]\!]\ \sigma \wedge \mathcal{B}[\![b_1]\!]\ \sigma$
$\mathcal{B}[\![\neg b]\!]\ \sigma = \neg(\mathcal{B}[\![b]\!]\ \sigma)$

Boolean – Implementation

```
valB :: Bexp -> State -> Bool

valB (True') state = True
valB (False') state = False
valB (Equal a1 a2) state = valA a1 state == valA a2 state
valB (LessEq a1 a2) state = valA a1 state <= valA a2 state
valB (And b1 b2) state = valB b1 state && valB b2 state
valB (Not b) state = not (valB b state)
```

Substitution – Intuition

$a_0[x \mapsto a]$

Replace all occurrences of the variable x in the expression a with the expression a_0. The operation is performed on the abstract representation. Using the concrete representation may require insertion of parentheses.

$$(x * y)[x \mapsto 3] = 3 * y$$
$$(x * y)[x \mapsto x + 1] = (x + 1) * y$$

Substitution – Definition

$$\begin{aligned}
n[x \mapsto a] &= n \\
x_1[x \mapsto a] &= \begin{cases} a & \text{if } x_1 = x \\ x_1 & \text{if } x_1 \neq x \end{cases} \\
(a_1 + a_2)[x \mapsto a] &= a_1[x \mapsto a] + a_2[x \mapsto a] \\
(a_1 * a_2)[x \mapsto a] &= a_1[x \mapsto a] * a_2[x \mapsto a] \\
(a_1 - a_2)[x \mapsto a] &= a_1[x \mapsto a] - a_2[x \mapsto a]
\end{aligned}$$

Substitution – Implementation

```
subst :: Aexp -> String -> Aexp -> Aexp
subst (Num n) _ _     = Num n
subst (Var x1) x a    = if x1==x then a else Var x1
subst (Add a1 a2) x a = Add (subst a1 x a) (subst a2 x a)
subst (Mul a1 a2) x a = Mul (subst a1 x a) (subst a2 x a)
subst (Sub a1 a2) x a = Sub (subst a1 x a) (subst a2 x a)
```