Edward Niemann

Hugh Morton

Ethan Christensen

Benjamin Sachs

Milestone 2

Section 1: Our Model of Computation

We will use the model m = (env, s) of a program state that we have developed. We will use functions access and getLoc to obtain the r-values of variables

We will define the meaning of program evaluation in terms of a set of semantic equations.

We will use let blocks to manage complexity of the right hand side of semantic equations.

Semantic equations will contain parse expressions.

We assume a computational context where a variety of semantic operations are given (or can be implemented).

Section 2: Definition of E’

We define a valuation function, named E’, that evaluates expression parse trees relative to a given program state.

E’ : parse\_expression \* model → value \* model

E’ will be defined as a set of equations.

We will write one equation for each grammar rule.

E’ ( [[ booleanOr1 ]], m ) = E’ ( booleanOr1, m )

E’ ( [[ booleanOr1 or booleanAnd1 ]], m0 ) =

let

val (v1, m1) = E’ ( booleanOr1, m0)

val (v2, m2) = E’ ( booleanAnd1, m1 )

in

(v1 or v2, m2)

end

E’ ( [[ booleanAnd1 ]], m ) = E’ ( booleanAnd1, m )

E’ ( [[ booleanAnd1 and booleanEquality1 ]], m0 ) =

let

val (v1, m1) = E’ ( booleanAnd1, m0)

val (v2, m2) = E’ ( booleanEquality1, m1 )

in

(v1 and v2, m2)

end

E’ ( [[ booleanEquality1 ]], m ) = E’ ( booleanEquality1, m )

E’ ( [[ booleanEquality1 != booleanComparison1 ]], m0 ) =

let

val (v1, m1) = E’ ( booleanEquality1, m0)

val (v2, m2) = E’ ( booleanComparison1, m1 )

in

(v1 != v2, m2)

end

E’ ( [[ booleanEquality1 == booleanComparison1 ]], m0 ) =

let

val (v1, m1) = E’ ( booleanEquality1, m0)

val (v2, m2) = E’ ( booleanComparison1, m1 )

in

(v1 == v2, m2)

end

E’ ( [[ booleanComparison1 ]], m ) = E’ ( booleanComparison1, m )

E’ ( [[ booleanComparison1 < addSubExpression1 ]], m0 ) =

let

val (v1, m1) = E’ ( booleanComparison1, m0)

val (v2, m2) = E’ ( addSubExpression1, m1 )

in

(v1 < v2, m2)

end

E’ ( [[ booleanComparison1 > addSubExpression1 ]], m0 ) =

let

val (v1, m1) = E’ ( booleanComparison1, m0)

val (v2, m2) = E’ ( addSubExpression1, m1 )

in

(v1 > v2, m2)

end

E’ ( [[ booleanComparison1 <= addSubExpression1 ]], m0 ) =

let

val (v1, m1) = E’ ( booleanComparison1, m0)

val (v2, m2) = E’ ( addSubExpression1, m1 )

in

(v1 <= v2, m2)

end

E’ ( [[ booleanComparison1 >= addSubExpression1 ]], m0 ) =

let

val (v1, m1) = E’ ( booleanComparison1, m0)

val (v2, m2) = E’ ( addSubExpression1, m1 )

in

(v1 >= v2, m2)

end

E’ ( [[ addSubExpression1 ]], m ) = E’ ( addSubExpression1, m )

E’ ( [[ addSubExpression1 + multDivModExpression1 ]], m0 ) =

let

val (v1, m1) = E’ ( addSubExpression1, m0)

val (v2, m2) = E’ ( multDivModExpression1, m1 )

in

(v1 + v2, m2)

end

E’ ( [[ addSubExpression1 - multDivModExpression1 ]], m0 ) =

let

val (v1, m1) = E’ ( addSubExpression1, m0)

val (v2, m2) = E’ ( multDivModExpression1, m1 )

in

(v1 - v2, m2)

end

E’ ( [[ multDivModExpression1 ]], m ) = E’ ( multDivModExpression1, m )

E’ ( [[ multDivModExpression1 \* unaryMinus1 ]], m0 ) =

let

val (v1, m1) = E’ ( multDivModExpression1, m0)

val (v2, m2) = E’ ( unaryMinus1, m1 )

in

(v1 \* v2, m2)

end

E’ ( [[ multDivModExpression1 / unaryMinus1 ]], m0 ) =

let

val (v1, m1) = E’ ( multDivModExpression1, m0)

val (v2, m2) = E’ ( unaryMinus1, m1 )

in

(v1 / v2, m2)

end

E’ ( [[ multDivModExpression1 mod unaryMinus1 ]], m0 ) =

let

val (v1, m1) = E’ ( multDivModExpression1, m0)

val (v2, m2) = E’ ( unaryMinus1, m1 )

in

(v1 mod v2, m2)

end

E’ ( [[ unaryMinus1 ]], m ) = E’ ( unaryMinus1, m )

E’ ( [[ - exponentExpression1 ]], m0 ) =

let

val (v1, m1) = E’ ( exponentExpression1, m0)

in

(v1 \* (-1), m1)

end

E’ ( [[ exponentExpression1]], m ) = E’ ( exponentExpression1, m )

E’ ( [[ logicNegation1 ^ exponentExpression1 ]], m0 ) =

let

val (v1, m1) = E’ ( exponentExpression1 , m0 )

val (v2, m2) = E’ ( logicNegation1, m1 )

in

(exp(v2, v1), m2)

end

E’ ( [[ logicNegation1]], m ) = E’ ( logicNegation1, m )

E’ ( [[ ! integerParenAbs1 ]], m0 ) =

let

val (v1, m1) = E’ ( integerParenAbs1, m0 )

in

(! v1, m1)

end

E’ ( [[ integerParenAbs1 ]], m ) = E’ (integerParenAbs1, m )

E’ ( [[ ( Expression1 ) ]], m ) = E’( Expression1, m )

E’ ( [[ | Expression1 | ]], m0 ) =

let

val (v1, m1 ) = E’ (Expression1, m0 )

in

(|v1|, m1)

end

E’ ( [[ integer ]], m ) = ( integer, m )

E’ ( [[ boolean ]], m ) = ( boolean, m )

E’ ( [[ identifier ]], m ) =

let

val loc = getLoc( accessEnv (identifier, m) )

val v = accessStore( loc, m )

in

(v, m)

end

E’ ( [[ DecoratedID1 ]], m ) = E’( DecoratedID1, m )

E’( [[ identifier ++ ]], m0 ) =

let

val loc = getLoc( accessEnv (identifier, m0) )

val v = accessStore( loc, m0 )

val m1 = updateStore ( loc, v + 1, m0 )

in

(v, m1)

end

E’( [[ ++ identifier ]], m0 ) =

let

val loc = getLoc( accessEnv (identifier, m0) )

val v = accessStore( loc, m0 )

val m1 = updateStore ( loc, v + 1, m0 )

in

(v + 1, m1)

end

E’( [[ identifier -- ]], m0 ) =

let

val loc = getLoc( accessEnv (identifier, m0) )

val v = accessStore( loc, m0 )

val m2 = updateStore ( loc, v - 1, m1 )

in

(v, m1)

end

E’( [[ -- identifier ]], m0 ) =

let

val loc = getLoc( accessEnv (identifier, m0) )

val v = accessStore( loc, m0 )

val m1 = updateStore ( loc, v - 1, m0 )

in

(v - 1, m1)

end

Section 3: Definition of M

M: parse\_expression \* model → model

M ( [[ StatementList1 ]], m0 ) = M ( StatementList1, m0)

M ( [[ ]], m ) = m

M ( [[ Statement1 StatementList1 ]], m0 ) =

let

val m1 = M( Statement1, m0 )

val m2 = M( StatementList1, m1 )

in

m2

end

M ( [[ Declaration1 ]], m ) = M ( Declaration1, m )

M ( [[ AssignmentStatement1 ]], m ) = M ( AssignmentStatement1, m )

M ( [[ DeclarationAssignment1 ]], m ) = M ( DeclarationAssignment1, m )

M ( [[ Conditional1 ]], m ) = M ( Conditional1, m )

M ( [[ Loop1 ]], m ) = M ( Loop1, m )

M ( [[ IncrementDecrement1 ]], m ) = M ( IncrementDecrement1, m )

M ( [[ Print1 ]], m ) = M ( Print1, m )

M ( [[ Block1 ]], m ) = M ( Block1, m )

M ( [[ int identifier ; ]], m0 ) =

let

val m1 = updateEnv( identifier, int, new(), m0 )

in

m1

end

M ( [[ boolean identifier ; ]], m0 ) =

let

val m1 = updateEnv( identifier, boolean, new(), m0 )

in

m1

end

M ( [[ Assignment ; ]], m ) = M ( Assignment, m )

M ( [[ identifier = Expression1 ]], m0 ) =

let

val (v, m1) = E’ ( Expression1, m0 )

val loc = getLoc ( accessEnv ( identifier, m1 ) )

val m2 = updateStore ( loc, v, m1 )

in

m2

end

M ( [[ boolean identifier = Expression1 ; ]], m0 ) =

let

val m1 = updateEnv( identifier, boolean, new(), m0 )

val (v, m2) = E’ ( Expression1, m1 )

val loc = getLoc ( accessEnv ( identifier, m2 ) )

val m3 = updateStore ( loc, v, m2 )

in

m3

end

M ( [[ int identifier = Expression1 ; ]], m0 ) =

let

val m1 = updateEnv( identifier, int, new(), m0 )

val (v, m2) = E’ ( Expression1, m1 )

val loc = getLoc ( accessEnv ( identifier, m2 ) )

val m3 = updateStore ( loc, v, m2 )

in

m3

end

M ( [[ if ( Expression1 ) Block1 ]], m0 ) =

let

val ( v, m1 ) = E’ ( Expression1, m0 )

in

if v then M ( Block1, m1 )

else m1

end

M ( [[ if ( Expression1 ) Block1 else Block2 ]], m0 ) =

let

val ( v, m1 ) = E’ ( Expression1, m0 )

in

if v then M ( Block1, m1 )

else M ( Block2, m1 )

end

M ( [[ while ( Expression1 ) Block1 ]], m ) = N ( Expression1, Block1, m )

M ( [[ for ( Assignment1 ; Expression1 ; LoopID1 ) Block1 ]], m0 ) =

let

val m1 = M ( Assignment1, m0 )

in

O ( Expression1, Block1, LoopID1, m1 )

end

M ( [[ DecoratedID1 ; ]], m0 ) =

let

val (v, m1) = E’ ( DecoratedID1, m0 )

in

m1

end

M ( [[ print ( Expression1 ) ; ]], m0 ) =

let

val (v, m1 ) = E’ ( Expression1, m0 )

in

print ( m1 )

end

M ( [[ { StatementList1 } ]], (env0, s0) ) =

let

val ( env1,s1 ) = M ( StatementList1, ( env0, s0 ) )

val m2 = ( env0, s1 )

in

m2

end

N: parse\_expression \* parse\_expression \* model → model

N ( Expression1, Block1, m0 ) =

let

val ( v, m1 ) = E’ ( Expression1, m0 )

in

if v then

let

val m2 = M ( Block1, m1 )

val m3 = N ( Expression1, Block1, m2 )

in

m3

end

else m1

end

O: parse\_expression \* parse\_expression \* parse\_expression \* model → model

O ( Expression1, Block1, LoopID1, m0 ) =

let

val ( v, m1 ) = E’ ( Expression1, m0 )

in

if v then

let

val m2 = M ( Block1, m1 )

val m3 = M ( LoopID1, m2 )

val m4 = N ( Expression1, Block1, m3 )

in

m4

end

else m1

end