# Program Transformation by Term Rewriting

Eelco Visser

Center for Software Technology

Utrecht University

visser@cs.uu.nl

September 27, 2002

## Program Transformation by Term Rewriting

- Symbolic computation: manipulating program representations instead of running programs
- Program transformation: modifying programs
- Term rewriting: replace subterms by other subterms
- Programs can be represented as terms (abstract syntax trees)
- Therefore: specify program transformations using rewrite rules
- Evidence: next slides provide some examples
  - Desugaring
  - Constant folding
  - Algebraic simplification
  - CoreHaskell simplifications

## Desugaring: Infix to Prefix

```
DefAnd   : [[ e1 & e2 ]]  -> [[ if e1 then e2 else 0 ]]
DefOr    : [[ e1 | e2 ]]  -> [[ if e1 then 1 else e2 ]]

DefUmin  : [[ - e ]]      -> [[ -(0, e) ]]
DefTimes : [[ e1 * e2 ]]  -> [[ *(e1, e2) ]]
DefDiv   : [[ e1 / e2 ]]  -> [[ /(e1, e2) ]]
DefPlus  : [[ e1 + e2 ]]  -> [[ +(e1, e2) ]]
DefMinus : [[ e1 - e2 ]]  -> [[ -(e1, e2) ]]

DefEq    : [[ e1 =  e2 ]] -> [[ =(e1, e2)  ]]
DefNeq   : [[ e1 <> e2 ]] -> [[ <>(e1, e2) ]]
DefGt    : [[ e1 >  e2 ]] -> [[ >(e1, e2)  ]]
DefLt    : [[ e1 <  e2 ]] -> [[ <(e1, e2)  ]]
DefGeq   : [[ e1 >= e2 ]] -> [[ >=(e1, e2) ]]
```

```
DefLeq   : [[ e1 <= e2 ]] -> [[ <=(e1, e2) ]]
```

## Sugaring: Prefix to Infix

```
BinTimes : [[ *(e1, e2) ]]  -> [[ e1 * e2 ]]
BinDiv   : [[ /(e1, e2) ]]  -> [[ e1 / e2 ]]
BinPlus  : [[ +(e1, e2) ]]  -> [[ e1 + e2 ]]
BinMinus : [[ -(e1, e2) ]]  -> [[ e1 - e2 ]]

BinEq    : [[ =(e1, e2)  ]] -> [[ e1 =  e2 ]]
BinNeq   : [[ <>(e1, e2) ]] -> [[ e1 <> e2 ]]
BinGt    : [[ >(e1, e2)  ]] -> [[ e1 >  e2 ]]
BinLt    : [[ <(e1, e2)  ]] -> [[ e1 <  e2 ]]
BinGeq   : [[ >=(e1, e2) ]] -> [[ e1 >= e2 ]]
BinLeq   : [[ <=(e1, e2) ]] -> [[ e1 <= e2 ]]
```

## Constant Folding: Evaluation

```
EvalAdd :
  [[ +(i, j) ]] -> [[ k ]]
  where <addS>(i, j) => k

EvalMin :
  [[ -(i, j) ]] -> [[ k ]]
  where <subtS> (i, j) => k

EvalMul :
  [[ *(i, j) ]] -> [[ k ]]
  where <mulS>(i, j) => k

EvalDiv :
  [[ /(i, j) ]] -> [[ k ]]
```

```
where <divS>(i, j) => k
```

## Constant Folding: Units and Zeros

```
AddZero :
  [[ +(0, e) ]] -> [[ e ]]
SubZero :
  [[ -(e, 0) ]] -> [[ e ]]
MulOne :
  [[ *(1, e) ]] -> [[ e ]]
DivOne :
  [[ /(e, 1) ]] -> [[ e ]]

MulZero :
  [[ *(0, e) ]] -> [[ 0 ]]
ZeroDiv :
  [[ /(0, e) ]] -> [[ 0 ]]
```

## Dead Code Elimination

```
EvalIf :
  [[ if 0 then e1 else e2 ]] -> [[ e2 ]]
EvalIf :
  [[ if i then e1 else e2 ]] -> [[ e1 ]]
  where <not(eq)> ([[ i ]], [[ 0 ]])
EvalIf :
  [[ if 0 then e ]] -> [[ () ]]
EvalIf :
  [[ if i then e ]] -> [[ e ]]
  where <not(eq)> ([[ i ]], [[ 0 ]])
EvalWhile :
  [[ while 0 do e ]] -> [[ () ]]
EvalFor :
  [[ for x := i to j do e ]] -> [[ () ]]
```

```
where <gtS>(i, j)
```

## Algebraic Simplification: Associativity and Commutativity

```
AddAssoc :
  [[ +(+(e1, e2), e3) ]] -> [[ +(e1, +(e2, e3)) ]]

AddComm :
  [[ +(e1, e2) ]] -> [[ +(e2, e1) ]]

AddCommInt :
  [[ +(e, i) ]] -> [[ +(i, e) ]]

MulAssoc :
  [[ *( *(e1, e2), e3) ]] -> [[ *(e1, *(e2, e3)) ]]

MulCommInt :
  [[ *(e, i) ]] -> [[ *(i, e) ]]
```

## Algebraic Simplification: Distributivity

```
DistMulAdd :
  [[ *(e1, +(e2, e3)) ]] -> [[ +( *(e1, e2), *(e1, e3) )

DistMulIntAdd :
  [[ *(i, +(e2, e3)) ]] -> [[ +( *(i, e2), *(i, e3) ) ]]

MulDiv :
  [[ *(i, /(e, j)) ]] -> [[ *(e, /(i, j)) ]]
```

## Algebraic Simplification: Let Bindings

```
LetSplit :
  [[ let d1 d2 ds in es end ]] ->
  [[ let d1
      in let d2 ds in es end
     end ]]

LetFlat  :
  [[ let ds1
     in let ds2 in es1 end;
        es2
    end ]] ->
  [[ let ds3 in es3 end ]]
  where <conc>(ds1, ds2) => ds3
      ; <conc>(es1, es2) => es3
```

## Algebraic Simplification: Sequences

```
SplitSeq :
  [[ (e1; e2; es) ]] -> [[ (e1; (e2; es)) ]]
  where <not([])> es
EmptySeq :
  [[ (); es) ]] -> [[ (es) ]]
SingletonSeq :
  [[ (e) ]] -> [[ e ]]
AssocSeqR :
  [[ ((e1; es1); es2) ]] -> [[ (e1; ((es1); es2)) ]]
  where <not([])> es2

FlatSeq :
  [[ (); es) ]] -> [[ (es) ]]
FlatSeq :
```

```
[[ (e; (es)) ]] -> [[ (e; es) ]]
```

## CoreHaskell Transformations

```
CaseOfKnownConstructor :
  [[ case C e* of { C x* -> e; alt* } ]] ->
  [[ parlet x* = e* in e ]]

CaseOfKnownConstructor :
  [[ case C1 e* of { C2 x* -> e; alt* } ]] ->
  [[ case C1 e* of { alt* } ]]
  where <not(eq)> (C1, C2)

AppOverCase :
  [[ e1 (case e2 of { alt1* }) ]] ->
  [[ case e2 of { alt2* } ]]
  where
    <map(\[[p -> e3]] -> [[p -> e1 e3]]\)> alt1* => alt2
```

## CoreHaskell Transformations

```
Beta :
  [[ (\ x* -> e) e* ]] ->
  [[ parlet x* = e* in e ]]

Inl :
  [[ let x = e1 in e2[x] ]] ->
  [[ let x = e1 in e2[~<rename>e1] ]]

Dead :
  [[ let x = e1 in e2 ]] -> [[ e2 ]]
  where <not(in)> (x, e2)
```

## CoreHaskell Transformations

```
LetFloat :
  [[ let x = (let y = e1 in e2) in e3 ]] ->
  [[ let y = e1
      in let x = e2
          in e3 ]]

Curry :
  [[ (e1 e* e2) ]] -> [[ (e1 e*) e2 ]]

UnCurry :
  [[ (e1 e*) e2 ]] -> [[ (e1 e* e2) ]]
```

# Java Generation

```
gen-read-reference:
  tuple@(oname, Reference(Name(name), Type(t@Object(tname))))
    ->
  [java:block-stm*[
      int ~id:idvar = resultSet.getInt(~string:idvar);

      ~storedtype ~localvar = null;
      try {
        ~id:localvar = _loader . ~id:getname ( ~id:idvar );
      } catch( LoadException exc ) {
        throw new SQLException("Foreign key exception: Could not load "
        + ~string:name + " (id: " + ~id:idvar +  ") related to this "
        + ~string:oname + " (id: " + resultSet.getInt("ID") + ")  Message: "
        + exc.getMessage());
      }
  ]]
    where <append-ID; localvar-name> name => idvar
```

```
; <localvar-name> name => localvar
; <get-method>    tname => getname
; <gen-stored-type> t  => storedtype
```

## Extensions of Standard Term Rewriting

Standard term rewriting has limitations

- Exhaustive, uncontrolled application of rules
- Rewrite rules are context-free

Stratego overcomes these limitations by means of

- Programmable rewriting strategies
- Dynamic rewrite rules