

# Dynamic Rewrite Rules

Eelco Visser

Utrecht University, The Netherlands  
[www.stratego-language.org](http://www.stratego-language.org)

Workshop on Higher-Order Rewriting (HOR'06)  
Seattle, August 15, 2006  
(Invited Talk)

joint work with  
Karina Olmos, Martin Bravenboer,  
Arthur van Dam, Bogdan Dumitriu

# Goal: Tools for Source-to-Source Transformation

## Transformations on various programming languages

- General-purpose languages
- (Embedded) domain-specific languages

## Combine different types of transformations

- Program generation and meta-programming
- Simplification
- (Domain-specific) optimization
- Data-flow transformations

## Source-to-source

- Transformations on abstract syntax trees

## Concise and reusable

## **Stratego/XT: language + tools for program transformation**

- XT: infrastructure for transformation systems
- Stratego: high-level language for program transformation
- Not tied to one type of transformation or language

## Stratego/XT: language + tools for program transformation

- XT: infrastructure for transformation systems
- Stratego: high-level language for program transformation
- Not tied to one type of transformation or language

## Stratego paradigm

- Rewrite rules for basic transformation steps
- Programmable rewriting strategies for controlling rules
- Dynamic rules for context-sensitive transformation
- Concrete syntax for patterns

## Stratego/XT: language + tools for program transformation

- XT: infrastructure for transformation systems
- Stratego: high-level language for program transformation
- Not tied to one type of transformation or language

## Stratego paradigm

- Rewrite rules for basic transformation steps
- Programmable rewriting strategies for controlling rules
- **Dynamic rules for context-sensitive transformation**
- Concrete syntax for patterns

# Program Transformation by Term Rewriting

Beta :

$\text{App}(\text{Lam}(x, e1), e2) \rightarrow \text{Let}(x, e2, e1)$

EvalAdd :

$\text{Add}(\text{Int}(i), \text{Int}(j)) \rightarrow \text{Int}(k)$

where  $\langle \text{add} \rangle (i, j) \Rightarrow k$

EvalIf :

$\text{If}(\text{False}(), e1, e2) \rightarrow e2$

- ASF+SDF
- innermost rewriting

# Program Transformation by Term Rewriting

Beta :

$\text{App}(\text{Lam}(x, e1), e2) \rightarrow \text{Let}(x, e2, e1)$

EvalAdd :

$\text{Add}(\text{Int}(i), \text{Int}(j)) \rightarrow \text{Int}(k)$

where  $\langle \text{add} \rangle (i, j) \Rightarrow k$

EvalIf :

$\text{If}(\text{False}(), e1, e2) \rightarrow e2$

- ASF+SDF
- innermost rewriting

## Applications

- typechecking
- interpretation
- grammar normalization
- parser generation
- compilation

# Problem I: Exhaustive application of rules

## Basic rewriting engines

- rewrite rules are applied until normal form is reached
- using standard strategy (e.g., innermost, outermost)

## 'Pure' rewrite rules often not confluent and/or terminating

- introduction of 'functions' to control application of rules
- induces many extra rules to implement traversal
- (or adapt the rewrite engine)

## Examples

- eager evaluation

`Eval(App(e1, e2)) -> Evalapp(App(Eval(e1), Eval(e2)))`

- lazy evaluation

`LazyEval(App(e1, e2)) -> Evalapp(App(LazyEval(e1), e2))`

# Strategies to Control Application of Rules

## Programmable rewriting strategies

- select rules to apply
- determine strategy to apply them with
- avoid overhead of traversal
- [Luttik & Visser 1997; Visser, Benaissa & Tolmach 1998]

## Ingredients

- strategy combinators ( $s_1; s_2$ ,  $s_1 <+ s_2$ ,  $id$ ,  $fail$ , ...)
- generic (parametric) strategy definitions
- one-step generic traversal operators ( $all$ ,  $one$ ,  $some$ , ...)
- generic traversal strategies

## Examples

```
topdown(s)  = s; all(topdown(s))  
bottomup(s) = all(bottomup(s)); s  
oncetd(s)   = s <+ one(oncetd(s))
```

## Inspired by strategies of ELAN [Kirchner et al. 1997]

- (ELAN did not support traversal strategies)

# Rewrite Rules and Strategies

## Constant folding

$y := x * (3 + 4) \implies y := x * 7$

## Constant folding rules

EvalAdd :  $[[ i + j ]] \rightarrow [[ k ]]$  where  $\langle \text{add} \rangle(i, j) \Rightarrow k$

EvalMul :  $[[ i * j ]] \rightarrow [[ k ]]$  where  $\langle \text{mul} \rangle(i, j) \Rightarrow k$

AddZero :  $[[ 0 + e ]] \rightarrow [[ e ]]$

## Constant folding strategy (bottom-up)

EvalBinOp = EvalAdd  $\leftarrow$  AddZero  $\leftarrow$  EvalMul  $\leftarrow$  EvalOther

try(s) = s  $\leftarrow$  id

constfold = all(constfold); try(EvalBinOp)

## Problem II: Context-Sensitive Transformations

### Rewrite rules are context-free

- Rewrite rules can only access information in term that is matched

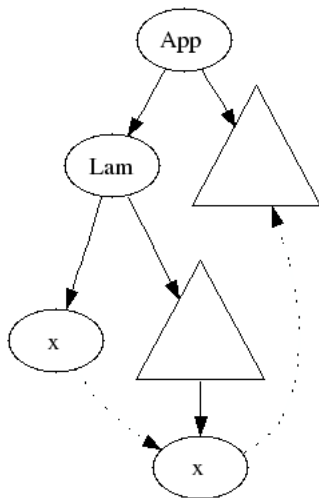
### Many transformations are context-sensitive

- Constant propagation
- Copy propagation
- Common-subexpression elimination
- Partial evaluation
- Function inlining
- Dead code elimination

# Terms with (Bound) Variables

## Concepts

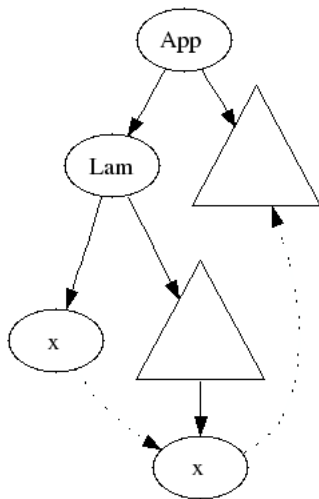
- variable declaration
- variable occurrence
- free variables
- variable scope



# Terms with (Bound) Variables

## Operations

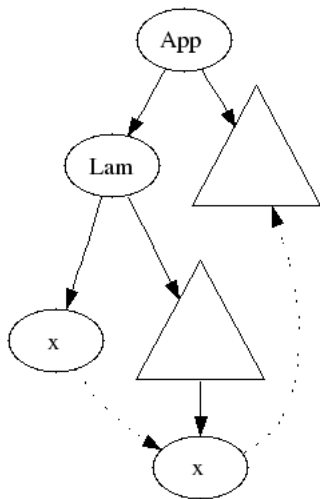
- rename bound variables
- substitute for free variables
- collect free variables
- equivalence
  - syntactical
  - modulo renaming of bound variables
  - equivalence modulo other laws (e.g.  $\beta$ ,  $\eta$ )
- generate programs
  - avoid variable capture
- optimize
  - reduce as much as possible
  - minimize code size



# Terms with (Bound) Variables

## Characteristics of lambda terms

- one variable per abstraction
- variable occurrences dominated by binder



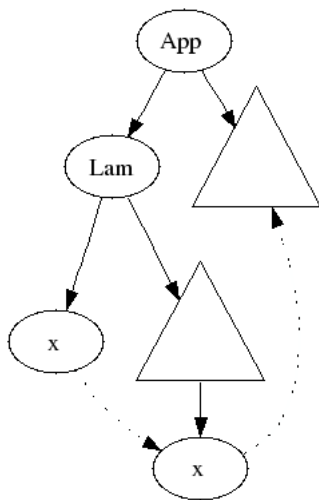
# Terms with (Bound) Variables

## Characteristics of lambda terms

- one variable per abstraction
- variable occurrences dominated by binder

## Many other 'binding' relations

- function definitions in C
- methods in Java
- assignments
- expression



# Terms with (Bound) Variables

## Characteristics of lambda terms

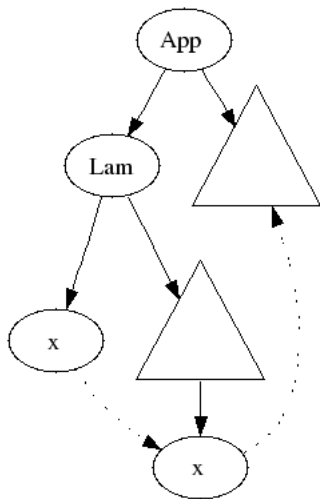
- one variable per abstraction
- variable occurrences dominated by binder

## Many other 'binding' relations

- function definitions in C
- methods in Java
- assignments
- expression

## Commonality

- connect information from context with occurrences
- not necessarily restricted to variables



## Functions in C

```
h();  
f() { h(); }  
g() { f(); }  
h() { g(); }
```

no definition before use

## Methods in Java

```
class A {  
    f() { ... }  
}  
class B {  
    A x;  
    g() { ... x.f() ... }  
}
```

no dominance relation

dynamic binding

# Constant Propagation

## Replace variable occurrences by their values

```
x := 1;  
a := ...;  
b := x + 1;  
x := f(a);  
...  
c := g(x);
```

multiple definitions  
of same variable

```
x := 1;  
if(...) {  
    x := 2  
}  
y := x + 1;
```

multiple bindings may reach  
same occurrence

# Dead Code Elimination

**Remove assignments the result of which is not used**

```
x := ...;    // live
a := ...;    // dead
b := x + 1;
x := f(a);   // dead
print(b);
```

binding is backward; use of variable  
keeps assignment alive

# Common Subexpression Elimination

**Replace expression with variable if computed before**

```
x := a + b;
```

```
c := ...;
```

```
y := a + b;
```

expression rather than variable is 'bound'

**How to extend rewriting  
to context-sensitive program transformation?**

## Represents programs as graphs

- make context information available locally
- by enriching the abstract syntax tree with extra links
- drawbacks
  - representation depends on transformation problem
  - transformation needs to preserve the representation
- exploration of strategic rewriting on (term) graphs in WRS'06
  - mixed term and graph representations
  - top-level structure of java programs as graphs, method bodies as DAGs

## Solution II: Contextual Rewrite Rules (ICFP'98)

### Rewrite at place where context information is available

- Appel & Jim (1997) Shrinking Lambda Expressions in Linear Time

UnfoldCall :

```
Let(FunDef(f, [x], e1), e2[f(e3)]) ->  
Let(FunDef(f, [x], e1), e2[Let(VarDef(x, e3), e1)])
```

### Problems

- only works if there is dominance relation
- replacement is hard to get right, unless knowledge of object language built into meta language
- expensive: local traversal to implement contextual rewriting

## Solution III: Dynamic Rewrite Rules

UnfoldCall :

```
Let(FunDef(f, [x], e1), e2) -> Let(FunDef(f, [x], e1), e3)
where <alltd((f(e3) -> Let(VarDef(x, e3), e1)))> e2 => e3
```

### Observation: contextual rule performs local rewrite

- local rewrite rule inherits variables from context
- local traversal (`alltd`) applies rewrite

DefineUnfoldCall =

```
?Let(FunDef(f, [x], e1), e2)
; rules( UnfoldCall : f(e3) -> Let(VarDef(x, e3), e1) )
```

### Dynamic rules

- Define a rewrite rule at place where context information is available and apply later
- dynamic rule inherits variable bindings from context
- multiple rules can be defined in a single traversal
- no extra local traversal is performed

# Defining and Undefining Rules Dynamically

## Constant Propagation and Folding in Straight-Line Code

```
b := 1;  
c := b + 3;  
b := foo();  
a := b + c
```

```
prop-const =  
  PropConst <- prop-const-assign  
  <- (all(prop-const); try(EvalBinOp))
```

```
prop-const-assign =  
  [[ x := <prop-const => e > ]]  
  ; if <is-value> e then  
    rules( PropConst : [[ x ]] -> [[ e ]] )  
  else  
    rules( PropConst :- [[ x ]] )  
  end
```

# Defining and undefining rules dynamically

## Constant Propagation and Folding in Straight-Line Code

```
b := 1;  
c := b + 3;  
b := foo();  
a := b + c
```

```
prop-const =  
  PropConst <- prop-const-assign  
  <- (all(prop-const); try(EvalBinOp))
```

```
prop-const-assign =  
  [[ x := <prop-const => e > ]]  
  ; if <is-value> e then  
    rules( PropConst : [[ x ]] -> [[ e ]] )  
  else  
    rules( PropConst :- [[ x ]] )  
  end
```

# Defining and Undefining Rules Dynamically

## Constant Propagation and Folding in Straight-Line Code

```
b := 1;  
c := b + 3;  
b := foo();  
a := b + c
```

```
prop-const =  
  PropConst ← prop-const-assign  
  ← (all(prop-const); try(EvalBinOp))
```

```
prop-const-assign =  
  [[ x := <prop-const => e > ]]  
  ; if <is-value> e then  
    rules( PropConst : [[ x ]] -> [[ e ]] )  
  else  
    rules( PropConst :- [[ x ]] )  
  end
```

# Defining and undefining rules dynamically

## Constant Propagation and Folding in Straight-Line Code

```
b := 1;  
c := b + 3;  
b := foo();  
a := b + c
```

```
b -> 1
```

```
prop-const =  
  PropConst <- prop-const-assign  
  <- (all(prop-const); try(EvalBinOp))
```

```
prop-const-assign =  
  [[ x := <prop-const => e > ]]  
  ; if <is-value> e then  
    rules( PropConst : [[ x ]] -> [[ e ]] )  
  else  
    rules( PropConst :- [[ x ]] )  
  end
```

# Defining and Undefining Rules Dynamically

## Constant Propagation and Folding in Straight-Line Code

```
b := 1;  
c := b + 3;  
b := foo();  
a := b + c
```

```
b -> 1
```

```
prop-const =  
  PropConst <- prop-const-assign  
  <- (all(prop-const); try(EvalBinOp))
```

```
prop-const-assign =  
  [[ x := <prop-const => e > ]]  
  ; if <is-value> e then  
    rules( PropConst : [[ x ]] -> [[ e ]] )  
  else  
    rules( PropConst :- [[ x ]] )  
  end
```

# Defining and Undefining Rules Dynamically

## Constant Propagation and Folding in Straight-Line Code

```
b := 1;  
c := b + 3;  
b := foo();  
a := b + c
```

```
b -> 1
```

```
prop-const =  
  PropConst <- prop-const-assign  
  <- (all(prop-const); try(EvalBinOp))
```

```
prop-const-assign =  
  [[ x := <prop-const => e > ]]  
  ; if <is-value> e then  
    rules( PropConst : [[ x ]] -> [[ e ]] )  
  else  
    rules( PropConst :- [[ x ]] )  
  end
```

# Defining and Undefining Rules Dynamically

## Constant Propagation and Folding in Straight-Line Code

```
b := 1;  
c := 1 + 3;  
b := foo();  
a := b + c
```

```
b -> 1
```

```
prop-const =  
  PropConst <- prop-const-assign  
  <- (all(prop-const); try(EvalBinOp))
```

```
prop-const-assign =  
  [[ x := <prop-const => e > ]]  
  ; if <is-value> e then  
    rules( PropConst : [[ x ]] -> [[ e ]] )  
  else  
    rules( PropConst :- [[ x ]] )  
  end
```

# Defining and Undefining Rules Dynamically

## Constant Propagation and Folding in Straight-Line Code

```
b := 1;  
c := 1 + 3;  
b := foo();  
a := b + c
```

```
b -> 1
```

```
prop-const =  
  PropConst <- prop-const-assign  
  <- (all(prop-const); try(EvalBinOp))
```

```
prop-const-assign =  
  [[ x := <prop-const => e > ]]  
  ; if <is-value> e then  
    rules( PropConst : [[ x ]] -> [[ e ]] )  
  else  
    rules( PropConst :- [[ x ]] )  
  end
```

# Defining and Undefining Rules Dynamically

## Constant Propagation and Folding in Straight-Line Code

```
b := 1;  
c := 4;  
b := foo();  
a := b + c
```

```
b -> 1
```

```
prop-const =  
  PropConst <- prop-const-assign  
  <- (all(prop-const); try(EvalBinOp))
```

```
prop-const-assign =  
  [[ x := <prop-const => e> ]]  
  ; if <is-value> e then  
    rules( PropConst : [[ x ]] -> [[ e ]] )  
  else  
    rules( PropConst :- [[ x ]] )  
  end
```

# Defining and Undefining Rules Dynamically

## Constant Propagation and Folding in Straight-Line Code

```
b := 1;  
c := 4;  
b := foo();  
a := b + c
```

```
b -> 1
```

```
prop-const =  
  PropConst <- prop-const-assign  
  <- (all(prop-const); try(EvalBinOp))
```

```
prop-const-assign =  
  [[ x := <prop-const => e > ]]  
  ; if <is-value> e then  
    rules( PropConst : [[ x ]] -> [[ e ]] )  
  else  
    rules( PropConst :- [[ x ]] )  
  end
```

# Defining and Undefining Rules Dynamically

## Constant Propagation and Folding in Straight-Line Code

```
b := 1;  
c := 4;  
b := foo();  
a := b + c
```

```
b -> 1 & c -> 4
```

```
prop-const =  
  PropConst <- prop-const-assign  
  <- (all(prop-const); try(EvalBinOp))
```

```
prop-const-assign =  
  [[ x := <prop-const => e > ]]  
  ; if <is-value> e then  
    rules( PropConst : [[ x ]] -> [[ e ]] )  
  else  
    rules( PropConst :- [[ x ]] )  
  end
```

# Defining and Undefining Rules Dynamically

## Constant Propagation and Folding in Straight-Line Code

```
b := 1;  
c := 4;  
b := foo();  
a := b + c
```

```
b -> 1 & c -> 4
```

```
prop-const =  
  PropConst <- prop-const-assign  
  <- (all(prop-const); try(EvalBinOp))
```

```
prop-const-assign =  
  [[ x := <prop-const => e > ]]  
  ; if <is-value> e then  
    rules( PropConst : [[ x ]] -> [[ e ]] )  
  else  
    rules( PropConst :- [[ x ]] )  
  end
```

# Defining and Undefining Rules Dynamically

## Constant Propagation and Folding in Straight-Line Code

```
b := 1;  
c := 4;  
b := foo();  
a := b + c
```

```
b - & c -> 4
```

```
prop-const =  
  PropConst <- prop-const-assign  
  <- (all(prop-const); try(EvalBinOp))
```

```
prop-const-assign =  
  [[ x := <prop-const => e > ]]  
  ; if <is-value> e then  
    rules( PropConst : [[ x ]] -> [[ e ]] )  
  else  
    rules( PropConst :- [[ x ]] )  
  end
```

# Defining and Undefining Rules Dynamically

## Constant Propagation and Folding in Straight-Line Code

```
b := 1;  
c := 4;  
b := foo();  
a := b + c
```

```
b - & c -> 4
```

```
prop-const =  
  PropConst <- prop-const-assign  
  <- (all(prop-const); try(EvalBinOp))
```

```
prop-const-assign =  
  [[ x := <prop-const => e > ]]  
  ; if <is-value> e then  
    rules( PropConst : [[ x ] -> [[ e ] ] )  
  else  
    rules( PropConst :- [[ x ] ] )  
  end
```

# Defining and Undefining Rules Dynamically

## Constant Propagation and Folding in Straight-Line Code

```
b := 1;  
c := 4;  
b := foo();  
a := b + 4
```

```
b - & c -> 4
```

```
prop-const =  
  PropConst <- prop-const-assign  
  <- (all(prop-const); try(EvalBinOp))
```

```
prop-const-assign =  
  [[ x := <prop-const => e > ]]  
  ; if <is-value> e then  
    rules( PropConst : [[ x ] -> [[ e ] ] )  
  else  
    rules( PropConst :- [[ x ] ] )  
  end
```

# Defining and Undefining Rules Dynamically

## Constant Propagation and Folding in Straight-Line Code

```
b := 1;  
c := 4;  
b := foo();  
a := b + 4
```

```
b - & c -> 4 & a -
```

```
prop-const =  
  PropConst <- prop-const-assign  
  <- (all(prop-const); try(EvalBinOp))
```

```
prop-const-assign =  
  [[ x := <prop-const => e > ]]  
  ; if <is-value> e then  
    rules( PropConst : [[ x ] -> [[ e ] ] )  
  else  
    rules( PropConst :- [[ x ] ] )  
  end
```

# Properties of Dynamic Rules

- Rules are defined dynamically
- Carry context information
- Multiple rules with same name can be defined
- Rules can be undefined
- Rules with same left-hand side override old rules

# Properties of Dynamic Rules

- Rules are defined dynamically
- Carry context information
- Multiple rules with same name can be defined
- Rules can be undefined
- Rules with same left-hand side override old rules

```
b := 3;  
...  
b := 4;
```

```
b -> 3  
b -> 3  
b -> 4
```

# Flow-Sensitive Transformations

## Flow-Sensitive Constant Propagation

```
(x := 3;  
y := x + 1;  
if foo(x) then  
  (y := 2 * x;  
   x := y - 2)  
else  
  (x := y;  
   y := 23);  
z := x + y)
```

```
(x := 3;  
y := 4;  
if foo(3) then  
  (y := 6;  
   x := 4)  
else  
  (x := 4;  
   y := 23);  
z := 4 + y)
```

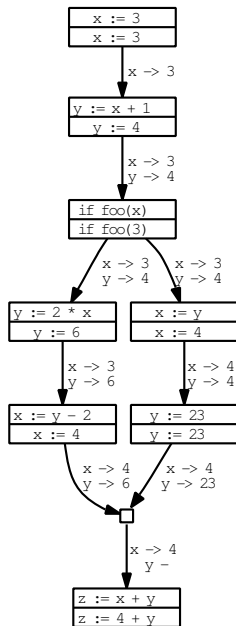
# Flow-Sensitive Transformations

## Flow-Sensitive Constant Propagation

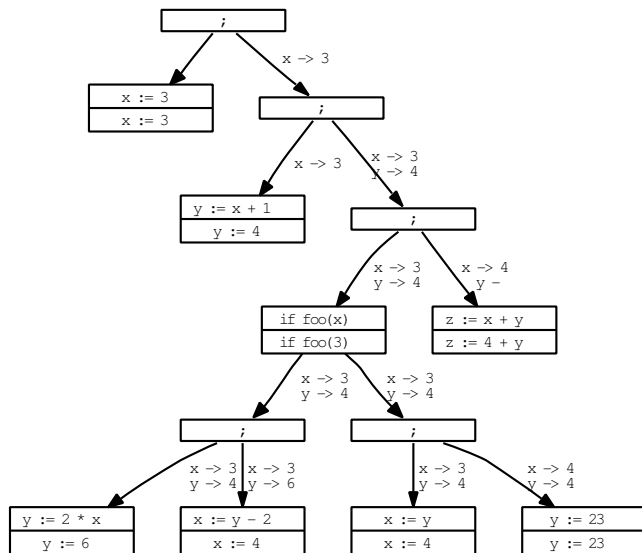
```
(x := 3;
 y := x + 1;
 if foo(x) then
   (y := 2 * x;
    x := y - 2)
 else
   (x := y;
    y := 23);
 z := x + y)
```

```
(x := 3;
 y := 4;
 if foo(3) then
   (y := 6;
    x := 4)
 else
   (x := 4;
    y := 23);
 z := 4 + y)
```

fork rule sets and combine at merge point



# Constant propagation in abstract syntax tree



# Forking and Intersecting Dynamic Rulesets

## Flow-sensitive Constant Propagation

```
prop-const-if =  
  [[ if <prop-const> then <id> else <id> ]]  
  ; ( [[if <id> then <prop-const> else <id>]]  
      /PropConst\ [[if <id> then <id> else <prop-const>]] )
```

$s_1$  /R\  $s_2$ : fork and intersect

# Propagation through Loops

```
(a := 1;  
i := 0;  
while i < m do (  
  j := a;  
  a := f();  
  a := j;  
  i := i + 1  
);  
print(a, i, j))
```

⇒

```
(a := 1;  
i := 0;  
while i < m do (  
  j := 1;  
  a := f();  
  a := 1;  
  i := i + 1  
);  
print(1, i, j))
```

## Flow-sensitive Constant Propagation

```
prop-const-while =  
  ?[[ while e1 do e2 ]]  
  ; (/PropConst)* [[while <prop-const> do <prop-const>]])
```

$/R^* s \equiv ((id /R\ s) /R\ s) /R\ \dots)$   
until fixedpoint of ruleset is reached

## Flow-sensitive Constant Propagation

```
prop-const-while =  
  ?[[ while e1 do e2 ]]  
  ; (/PropConst)* [[while <prop-const> do <prop-const>]])
```

$/R^* s \equiv ((id /R\ s) /R\ s) /R\ \dots)$   
until fixedpoint of ruleset is reached

prop-const-while terminates:  
fewer rules defined each iteration

# Combining Analysis and Transformation

## Unreachable code elimination

```
i := 1;  
j := 2;  
if j = 2  
  then i := 3;  
  else z := foo()  
print(i)
```

⇒

```
i := 1;  
j := 2;  
i := 3;  
print(3)
```

# Combining Analysis and Transformation

## Unreachable code elimination

```
i := 1;  
j := 2;  
if j = 2  
  then i := 3;  
  else z := foo()  
print(i)
```

⇒

```
i := 1;  
j := 2;  
i := 3;  
print(3)
```

```
EvalIf : [[ if 0 then e1 else e2 ]] -> [[ e2 ]]  
EvalIf : [[ if i then e1 else e2 ]] -> [[ e1 ]]  
        where <not(eq)>(i, [[0]])
```

# Combining Analysis and Transformation

## Unreachable code elimination

```
i := 1;  
j := 2;  
if j = 2  
  then i := 3;  
  else z := foo()  
print(i)
```

⇒

```
i := 1;  
j := 2;  
i := 3;  
print(3)
```

```
EvalIf : [[ if 0 then e1 else e2 ]] -> [[ e2 ]]  
EvalIf : [[ if i then e1 else e2 ]] -> [[ e1 ]]  
        where <not(eq)>(i, [[0]])
```

```
prop-const-if =  
  [[ if <prop-const> then <id> else <id> ]];  
(EvalIf; prop-const  
  <- ([[if <id> then <prop-const> else <id>]] /PropConst\  
      [[if <id> then <id> else <prop-const>]]))
```

# Combining Analysis and Transformation

## Unreachable code elimination

```
(x := 10;
 while A do
   if x = 10
     then dosomething()
     else (dosomethingelse();
           x := x + 1);
 y := x)
```

⇒

```
(x := 10;
 while A do
   dosomething();
 y := 10)
```

Conditional Constant Propagation [Wegman & Zadeck 1991]  
Graph analysis + transformation in Vortex [Lerner et al. 2002]

# Scope Labels – Constant Propagation with Local Variables

```
let var a := 1 var b := 2 var c := 3
  in a := b + c;
    let var c := a + 1
      in b := b + c;
        a := a + b;
        b := z + b end;
    a := c + b + a end
```

↓

```
let var a := 1 var b := 2 var c := 3
  in a := 5;
    let var c := 6
      in b := 8;
        a := 13;
        b := z + 8 end;
    a := 3 + b + 13 end
```

# Constant Propagation with Local Variables

```
prop-const = PropConst <+ prop-const-assign  
  <+ prop-const-let <+ prop-const-vardec  
  <+ all(prop-const); try(EvalBinOp <+ EvalRelOp)
```

```
prop-const-let =  
  |[ let <*id> in <*id> end ]|  
  ; {| PropConst : all(prop-const) |}
```

```
prop-const-vardec =  
  |[ var x ta := <prop-const => e> ]|  
  ; if <is-value> e  
    then rules( PropConst+x : |[ x ]| -> |[ e ]| )  
    else rules( PropConst+x :- |[ x ]| ) end
```

```
prop-const-assign =  
  |[ x := <prop-const => e> ]|  
  ; if <is-value> e  
    then rules( PropConst.x : |[ x ]| -> |[ e ]| )  
    else rules( PropConst.x :- |[ x ]| ) end
```

# Putting it all together

## Conditional Constant Propagation

```
prop-const =
  PropConst <+ prop-const-assign <+ prop-const-declare
  <+ prop-const-let <+ prop-const-if <+ prop-const-while
  <+ (all(prop-const); try(EvalBinOp))

prop-const-assign =
  [[ x := <prop-const => e > ]]
  ; if <is-value> e then rules( PropConst.x : [[ x ]] -> [[ e ]] )
    else rules( PropConst.x :- [[ x ]] ) end

prop-const-declare =
  [[ var x := <prop-const => e > ]]
  ; if <is-value> e then rules( PropConst+x : [[ x ]] -> [[ e ]] )
    else rules( PropConst+x :- [[ x ]] ) end

prop-const-let =
  ?[[ let d* in e* end ]]; { | PropConst : all(prop-const) | }

prop-const-if =
  [[ if <prop-const> then <id> else <id> ]]
  ; (EvalIf; prop-const
    <+ ([[ if <id> then <prop-const> else <id> ]]
        /PropConst \ [[ if <id> then <id> else <prop-const> ]]))

prop-const-while =
  ?[[ while e1 do e2 ]]
  ; ([[ while <prop-const> do <id> ]]; EvalWhile
    <+ (/PropConst \ \ [[ while <prop-const> do <prop-const> ]]))
```

# Recapitulation

- Rewrite rules for constant folding
- Strategies for (generic) traversal
- Dynamic rule propagates values
- Fork and intersection (union) for flow-sensitive transformation
- Dynamic rule scopes controls lifetime of rules

# Recapitulation

- Rewrite rules for constant folding
- Strategies for (generic) traversal
- Dynamic rule propagates values
- Fork and intersection (union) for flow-sensitive transformation
- Dynamic rule scopes controls lifetime of rules

can this be applied to other data-flow transformations?

# Copy Propagation

Replace copies  $x$  produced by assignments of the form  $x := y$  by original  $y$

```
a := b;  
c := d + a
```

```
a := b;  
c := d + b
```

# Copy Propagation

Replace copies  $x$  produced by assignments of the form  $x := y$  by original  $y$

```
a := b;  
c := d + a
```

```
a := b;  
c := d + b
```

## First attempt using dynamic rules (wrong)

```
copy-prop-assign =  
  ?|[ x := y ]|;  
  if <not(eq)>(x,y) then  
    rules( CopyProp.x : |[ x ]| -> |[ y ]| )  
  else  
    rules( CopyProp.x :- |[ x ]| )  
  end
```

## Problem: Insufficient Dependencies

```
(a := b;  
b := foo();  
c := d + a)
```

```
(a := b;  
b := foo();  
c := d + b)
```

```
copy-prop-assign =  
  ?|[ x := y ]|;  
  if <not(eq)>(x,y) then  
    rules( CopyProp.x : |[ x ]| -> |[ y ]| )  
  else  
    rules( CopyProp.x :- |[ x ]| )  
  end
```

# Problem: Insufficient Dependencies

```
(a := b;  
 b := foo();  
 c := d + a)
```

```
(a := b;  
 b := foo();  
 c := d + b)
```

- Problem: rule not undefined when variable in rhs changed
- Solution: undefine rule when any of its variables is modified

```
copy-prop-assign =  
  ?|[ x := y ]|;  
  if <not(eq)>(x,y) then  
    rules( CopyProp.x : |[ x ]| -> |[ y ]| )  
  else  
    rules( CopyProp.x :- |[ x ]| )  
  end
```

## Problem: Free Variable Capture

```
let var a := bar()
    var b := baz()
  in a := b;
    let var b := foo()
      in print(a)
    end
end
```

```
let var a := bar()
    var b := baz()
  in a := b;
    let var b := foo()
      in print(b) // wrong!
    end
end
```

```
copy-prop-assign =
  ?|[ x := y ]|;
  if <not(eq)>(x,y) then
    rules( CopyProp.x : |[ x ]| -> |[ y ]| )
  else
    rules( CopyProp.x :- |[ x ]| )
  end
```

# Problem: Free Variable Capture

```
let var a := bar()
    var b := baz()
  in a := b;
    let var b := foo()
      in print(a)
    end
end
```

```
let var a := bar()
    var b := baz()
  in a := b;
    let var b := foo()
      in print(b) // wrong!
    end
end
```

- Problem: rule not undefined when variables become shadowed
- Solution: undefine rule locally when some variable shadowed

```
copy-prop-assign =
  ?|[ x := y ]|;
  if <not(eq)>(x,y) then
    rules( CopyProp.x : |[ x ]| -> |[ y ]| )
  else
    rules( CopyProp.x :- |[ x ]| )
  end
```

## Problem: Escaping Variables (1)

```
let var a := bar()
  in let var b := foo()
      in a := b
      end;
  print(a)
end
```

```
let var a := bar()
  in let var b := foo()
      in a := b
      end;
  print(b) // wrong!
end
```

```
copy-prop-assign =
  ?|[ x := y ]|;
  if <not(eq)>(x,y) then
    rules( CopyProp.x : |[ x ]| -> |[ y ]| )
  else
    rules( CopyProp.x :- |[ x ]| )
  end
```

## Problem: Escaping Variables (1)

```
let var a := bar()
  in let var b := foo()
    in a := b
    end;
  print(a)
end
```

```
let var a := bar()
  in let var b := foo()
    in a := b
    end;
  print(b) // wrong!
end
```

- Problem: rule not undefined when a variable goes out of scope
- Solution: (re)define rule in local scope

```
copy-prop-assign =
  |[ x := y ]|;
  if <not(eq)>(x,y) then
    rules( CopyProp.x : |[ x ]| -> |[ y ]| )
  else
    rules( CopyProp.x :- |[ x ]| )
  end
```

## Problem: Escaping Variables (2)

```
let var a := bar()
    var c := baz()
  in let var b := foo()
      in a := b;
        a := c
      end;
    print(a)
  end
```

```
let var a := bar()
    var c := baz()
  in let var b := foo()
      in a := b;
        a := c
      end;
    print(c) // ok!
  end
```

```
copy-prop-assign = ?|[ x := y ]|;
  if <not(eq)>(x,y) then
    rules( CopyProp.x : |[ x ]| -> |[ y ]| )
  else rules( CopyProp.x :- |[ x ]| ) end
```

## Problem: Escaping Variables (2)

```
let var a := bar()
    var c := baz()
  in let var b := foo()
      in a := b;
        a := c
      end;
    print(a)
  end
```

```
let var a := bar()
    var c := baz()
  in let var b := foo()
      in a := b;
        a := c
      end;
    print(c) // ok!
  end
```

- Problem: definition in local scope is too restricted
- Solution: (re)define rule in *innermost* scope of all variables involved

```
copy-prop-assign = ?|[ x := y ]|;  
if <not(eq)>(x,y) then  
  rules( CopyProp.x : |[ x ]| -> |[ y ]| )  
else rules( CopyProp.x :- |[ x ]| ) end
```

## Dependent Dynamic Rules

- Declare rule dependencies

```
R.lab : p1 -> p2
      depends on [(lab1,dep1), ..., (labn,depn)]
```

- Undefine all rules depending on dep

```
undefine-R(|dep)
```

- Locally undefine all rules depending on dep

```
new-R(|lab, dep)
```

and label current scope with lab

```
copy-prop-assign = ?|[ x := y ]|
; undefine-CopyProp(|x)
; if <not(eq)>(x,y) then
  rules(
    CopyProp.x : |[ x ]| -> |[ y ]|
    depends on [(x,x),(y,y)]
  )
end
```

# Common-Subexpression Elimination

```
(x := a + b;  
 y := a + b;  
 z := a + c;  
 a := 1;  
 z := (a + c) + (a + b))
```

$\Rightarrow$

```
(x := a + b;  
 y := x;  
 z := a + c;  
 a := 1;  
 z := (a + c) + (a + b))
```

# Common-Subexpression Elimination

```
(x := a + b;  
y := a + b;  
z := a + c;  
a := 1;  
z := (a + c) + (a + b))
```

$\Rightarrow$

```
(x := a + b;  
y := x;  
z := a + c;  
a := 1;  
z := (a + c) + (a + b))
```

## Assignment

$x := e$

## Propagation rule

$|[ e ]| \rightarrow |[ x ]|$

## Dependencies in common-subexpression elimination

- all variables in assignment  $x := e$

# CSE with Dependent Dynamic Rules

```
cse-assign =  
  [[ x := <cse => e > ]]  
  ; where( undefine-CSE(|x|) )  
  ; where( <pure-and-not-trivial(|x|)> e )  
  ; where( get-var-dependencies => xs )  
  ; rules( CSE : [[ e ] -> [[ x ] depends on xs )
```

```
cse-if =  
  [[ if <cse> then <id> else <id> ]]  
  ; ( [[ if <id> then <cse> else <id> ]  
      /CSE\ [[ if <id> then <id> else <cse> ]])
```

```
cse-while =  
  [[ while <id> do <id> ]]  
  ; (/CSE\* [[ while <cse> do <cse> ]])
```

# Recapitulation

- Rewrite rules for basic transformations
- Strategies for control and (generic) traversal
- Dynamic rule propagates context information
- Fork and intersection (union) for flow-sensitive transformation
- Dynamic rule scopes and **dependent rules** for control over lifetime of rules

## Examples

- Constant propagation  $|[ x ]| \rightarrow |[ i ]|$
- Copy propagation  $|[ x ]| \rightarrow |[ y ]|$
- Common-subexpression elimination  $|[ e ]| \rightarrow |[ x ]|$
- Forward substitution  $|[ x ]| \rightarrow |[ e ]|$
- Partial redundancy elimination (down-safe, earliest)

# Generic Data-Flow Strategies

- Observation: most of data-flow strategy is boilerplate
- Solution: generic data-flow strategy
- Generalized operators
  - Intersection and union:  $/R_{s_1} \setminus R_{s_2}/$  and  $/R_{s_1} \setminus R_{s_2}/^*$
  - Undefined of multiple dynamic rules

## Instantiation for common-subexpression elimination

```
cse = forward-prop(fail, id, cse-after | ["CSE"], [], [])
```

```
cse-assign =  
  ?[ [ x := e ]  
    ; where( <pure-and-not-trivial(|x)> [ e ] )  
    ; where( get-var-dependencies => xs )  
    ; rules( CSE : [ e ] -> [ x ] depends on xs )
```

```
cse-after = try(cse-assign <+ CSE)
```

# Combining Transformations

```
super-opt =  
  forward-prop(  
    prop-const-transform  
    , bvr-before  
    , bvr-after; copy-prop-after  
      ; prop-const-after; cse-after  
    | ["PropConst", "CopyProp", "CSE"]  
    , []  
    , ["RenameVar"]  
  )
```

Apply multiple data-flow transformations simultaneously

# Experience with Dynamic Rules

- Tiger compiler: sandbox for transformation techniques  
bound variable renaming, inlining, constant propagation, copy propagation, common-subexpression elimination, dead assignment elimination, partial redundancy elimination, online and offline partial evaluation, loop normalization, loop vectorization, ...
- Octave compiler  
type specialization, partial evaluation, other data-flow transformations, combined transformations, loop vectorization
- Stratego compiler  
inlining, specialization, bound-unbound variables analysis, ...
- LVM optimizer (functional)  
substitutions, inlining, (deforestation, warm fusion)
- Java Compiler  
name disambiguation, type propagation, assimilation of embedded domain-specific languages

### [Wegman & Zadeck 1991]

- SCC: special algorithm for conditional constant propagation
- propagation through SSA edges

### [Lerner et al. 2002]

- integration of analysis and transformation for CFGs
- combination of multiple analyses/transformations

### [Lacey & de Moor 2001]

- temporal logic : find context from occurrence

### [Sittampalam, de Moor & Larsen 2004]

- regular path queries
- incremental analysis after applying transformation

## Interprocedural transformation

- Type specialization for Octave [Olmos & Visser 2003]
- Poly-variant online specialization and unfolding [Bravenboer, Van Dam, Olmos & Visser 2005]
- Global variables
- Mono-variant specialization (summaries)

## Aliasing

- Propagation with records and arrays

## Control-flow

- support for other control-flow constructs [Dumitriu, 2006]

# Conclusion

- Dynamic rewrite rules
  - rewriting for context-sensitive program transformations
- Abstract interpretation style of data-flow transformation
  - combination of data-flow analysis and transformation
- Hygienic: correct treatment of variable binding constructs
  - avoid free variable capture and escaping variables
  - scoped transformation rules
- Generic data-flow strategies
  - concise specification of data-flow transformation
  - combination of multiple transformations
- Combination of data-flow transformations with other transformations
  - reuse of (elements of) transformations
  - alternative transformation strategies

The End