Extensible Program Transformation Systems

Eelco Visser

Institute of Information & Computing Sciences
Utrecht University,
The Netherlands

March 6, 2005
IFIP WG 2.11 meeting Rice University, Houston

Mission Statement of the Stratego/XT Project

To design a language and toolset for expressing all aspects of all kinds of program transformations for all kinds of languages, concisely and efficiently.

Transformations

- Simplification
- Optimization
  - data-flow transformation
  - inlining
  - partial evaluation
  - vectorization
  - ...
- Assimilation
- Generation
- Compilation
- Refactoring
- Documentation generation
- ...

Languages

- Procedural
- Functional
- Object-oriented
- Logic
- Anything with structured representation

Implementation Aspects

- Syntax definition (SDF)
- Tree representation (ATerm)
- Pretty-printing (GPP)
- Transformation strategies (Stratego)
- Transformation tool composition (XTC)

Research Directions

1. Universal abstractions for program transformation

Research Directions

1. Universal abstractions for program transformation
2. Domain-specific language embedding

Research Directions

1. Universal abstractions for program transformation
2. Domain-specific language embedding
3. Extensible transformation systems

Research Directions

1. Universal abstractions for program transformation
2. Domain-specific language embedding
3. Extensible transformation systems
4. Meta-programming by program transformation
   (not discussed now)

Stratego: universal abstractions for program transformation

Stratego: universal abstractions for program transformation

Core abstractions

- Matching and building terms
- Sequential composition and choice
- One-level traversal operators (specific and generic)
- Dynamic rules (context-sensitive rewriting)

Stratego: universal abstractions for program transformation

Core abstractions

- Matching and building terms
- Sequential composition and choice
- One-level traversal operators (specific and generic)
- Dynamic rules (context-sensitive rewriting)

Derived abstractions

- Match-build combinations
  - rewrite rules, term wrap, ...
- Full tree traversals
- Generic strategies

Constant Folding

```
EvaPlus :
  Plus(Int(i), Int(j)) -> Int(k)
  where <add>(i,j) => k

EvaMul :
  Mul(Int(i),  Int(j)) -> Int(k)
  where <mul>(i,j) => k

EvalBinOp =
  EvalPlus <+ EvalMul <+ ...

const-fold =
  all(const-fold); try(EvalBinOp)
```

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)
```

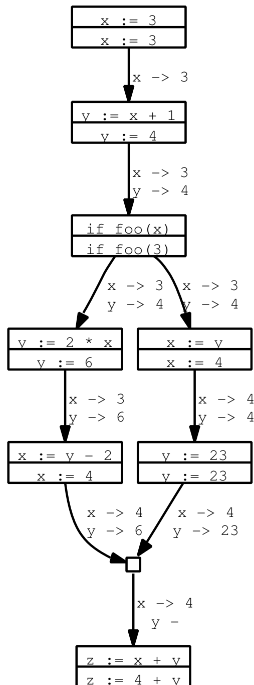
```
(x := 3;
 y := 4;
 if foo(3) then
   (y := 6;
    x := 4)
 else
   (x := 4;
    y := 23);
 z := 4 + y)
```

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)
```

using dynamic rewrite rules



Constant Propagation Strategy

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

prop-const-assign =
  Assign(?x, prop-const => e)
  ; if <is-value> e
    then rules( PropConst :  Var(x) -> e )
    else rules( PropConst :- Var(x) ) end

prop-const-if =
 If(prop-const, id, id)
  ; If(id,prop-const,id) /PropConst\ If(id,id,prop-const)

prop-const-while =
  ?While(e1, e2)
  ; (/PropConst\* While(prop-const, prop-const))
```

Generic Data-Flow Transformation Strategy

```
forward-prop(transform, before, after | Rs) =
let fp = prop-assign <+ prop-if <+ prop-while
         <+ transform(fp) <+ (before; all(fp); after)

    prop-assign =
      Assign(id, fp)
      ; (transform(fp)
         <+ before; ?Assign(Var(x), e)
            ; undefine-dynamic-rules(|Rs,x); after)

    prop-if =
      If(fp, id, id)
      ; (transform(fp) <+
         before; (If(id,fp,id) /~Rs\ If(id,id,fp)); after)
    ...
 in fp end
```

Common-Subexpression Elimination Strategy

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

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

```
(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 Strategy

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

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

Record dependencies of dynamic rules

```
(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))
```

Domain-specific language embedding

MetaBorg

- Embedding (domain-specific) language in host language
- Assimilation of embedded code with host code

General architecture

- Modular syntax definition with SDF
- Assimilation with Stratego
- Combine arbitrary languages

Concrete Syntax in Stratego

Use concrete syntax of object language in term patterns

```
Normalize :
 |[ for x := e1 to e2 do e3 ]| ->
    |[ for y := 1 to (e2 - e1) + 1 do e3 ]|
 where <not(?|[ 1 ]|)> e1
     ; new => y
     ; rules( ReplIV : |[ x ]| ->  |[ y + e1 - 1 ]| )
```

Concrete Syntax in Stratego

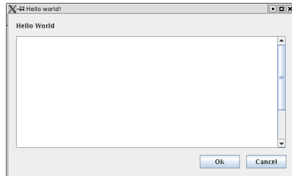
Use concrete syntax of object language in term patterns

```
Normalize :
 |[ for x := e1 to e2 do e3 ]| ->
    |[ for y := 1 to (e2 - e1) + 1 do e3 ]|
 where <not(?|[ 1 ]|)> e1
     ; new => y
     ; rules( ReplIV : |[ x ]| ->  |[ y + e1 - 1 ]| )
```

```
Normalize :
 For(Var(x), e1, e2, e3) ->
   For(Var(y), Int("1"), Plus(Minus(e2, e1),Int("1")), e3)
 where <not(?Int("1"))> e1; new => y
     ; rules(
         ReplIV : Var(x) -> Minus(Plus(Var(y),e1),Int("1"))
       )
```

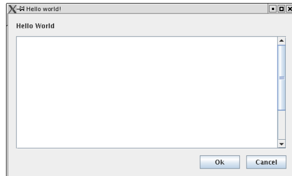
```
public class HelloWorld {
  public static void main(String[] ps) {

    JPanel panel = panel of border layout {
      north = label "Hello World"

      center = scrollpane of textarea {
        rows    = 20
        columns = 40
      }

      south = panel of border layout {
        east = panel of grid layout {
          row = {
            button "Ok"
            button "Cancel"
          }
        }
      }
    };

    ...
```

JavaSwul



```
public class HelloWorld {
public static void main(String[] ps) {

 JTextArea text = new JTextArea(20,40);

 JPanel panel
        = new JPanel(new BorderLayout(12,12));
 panel.add(BorderLayout.NORTH,
                    new JLabel("Hello World"));
 panel.add(BorderLayout.CENTER,
                    new JScrollPane(text));
 JPanel south
        = new JPanel(new BorderLayout(12,12));
 JPanel buttons
     = new JPanel(new GridLayout(1, 2, 12, 12));
 buttons.add(new JButton("Ok"));
 buttons.add(new JButton("Cancel"));

 south.add(BorderLayout.EAST, buttons);
 panel.add(BorderLayout.SOUTH, south);
 ...
```

JavaSwul



Extensible Transformation Systems: Use Cases

- Extend language
- Add transformation component

Extensible Transformation Systems: Issues

- Language extension requires extension of all transformations
- Anticipated vs unanticipated extension
- Concurrent extensions
- Design mismatch

Challenge

- A generic approach for extending transformation systems (instead of an extensible compiler for a specific language)
- What are the universal abstractions of extensibility?

Current Extension Mechanisms in Stratego/XT

- Transformation plugins
- Syntax plugins
- Parameterized strategies
- Extending definitions
- Adaptation with aspects (experimental)

Parameterized Strategies

Advantages

- Recombine reusable items
- Concurrent extensions

Disadvantages

- Anticipate variation points and lots of parameters
- No reuse of composition
- May not match needs of extension

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

Extending Definitions

- Advantage: no parameter overhead, no recomposition
- Disadvanteg: no concurrent extensions

Base

```
typecheck = bottomup(try(typecheck-rules))
typecheck-rules = typecheck-if <+ typecheck-plus
typecheck-plus :
  |[ (e1 :: Int) + (e2 :: Int) ]| -> |[ (e1 + e2) :: Int ]|
```

Extension

```
typecheck-rules = typecheck-panel
typecheck-panel :
  |[ panel of l { fld* } ]| ->
  |[ panel of l { fld* } :: JPanel ]|
  where ...
```

Adaptation with Aspects

```
module forward-prop-usage
imports forward-prop
aspects
 aspect prop-const =
  around : prop-rule(r) = proceed(["PropConst"])
  around : prop-rule = PropConst
  around : forward-prop = (proceed; try(EvalBinOp))
  before : prop-assign-ext =
             ?Assign(Var(x),e)
             ; if <is-value> e
               then rules(PropConst: Var(x) -> e)
               else rules(PropConst:- Var(x)) end
  around : prop-if = EvalIf; forward-prop <+ proceed
  around : prop-while =
             (While(forward-prop,id); EvalWhile <+ proceed)
```

Summary

- Concise specification of transformation strategies
- Easy extension of languages
- How to integrate these?

Summary

- Concise specification of transformation strategies
- Easy extension of languages
- How to integrate these?

What are the universal abstractions of extensibility?