A Brief Stratego/XT Tutorial

Karl Trygve Kalleberg
Eelco Visser

Dagstuhl Workshop: Beyond Program Slicing, 7.-11. Nov 2005

Plan of Attack

- Explain the basics of Stratego
  - *Terms*, *signatures*, *rewrite rules*, *strategies*, *concrete syntax*, *dynamic rules*
- Motivate with a small case-study
  - Constant propagation
- Give a tiny tool demo
- Take questions

Outline

Part I

Motivation

Example: Desugaring

```
for i := 1 to n do
  for j := 1 to n do
    c[i,j] := sum k = 1 to n (a[i,k] * b[k,j])
```

Example: Desugaring

```
for i := 1 to n do
  for j := 1 to n do
    c[i,j] := sum k = 1 to n (a[i,k] * b[k,j])
```

```
for i := 1 to n do
  for j := 1 to n do
    c[i,j] := let var d := 0
              in for k := 1 to n do
                   d := d + a[i,k] * b[k,j];
                 d
              end
```

Example: Desugaring

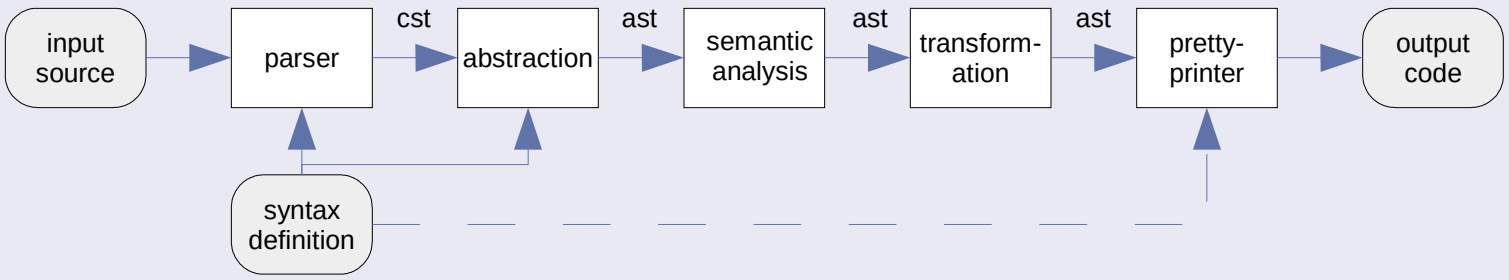
```
for i := 1 to n do
  for j := 1 to n do
    c[i,j] := sum k = 1 to n (a[i,k] * b[k,j])
```

```
for i := 1 to n do
  for j := 1 to n do
    c[i,j] := let var d := 0
              in for k := 1 to n do
                   d := d + a[i,k] * b[k,j];
                 d
              end
```

```
let var d
 in for i := 1 to n do
      for j := 1 to n do
        (d := 0;
         for k := 1 to n do d := d + a[i,k] * b[k,j];
         c[i,j] := d)
end
```

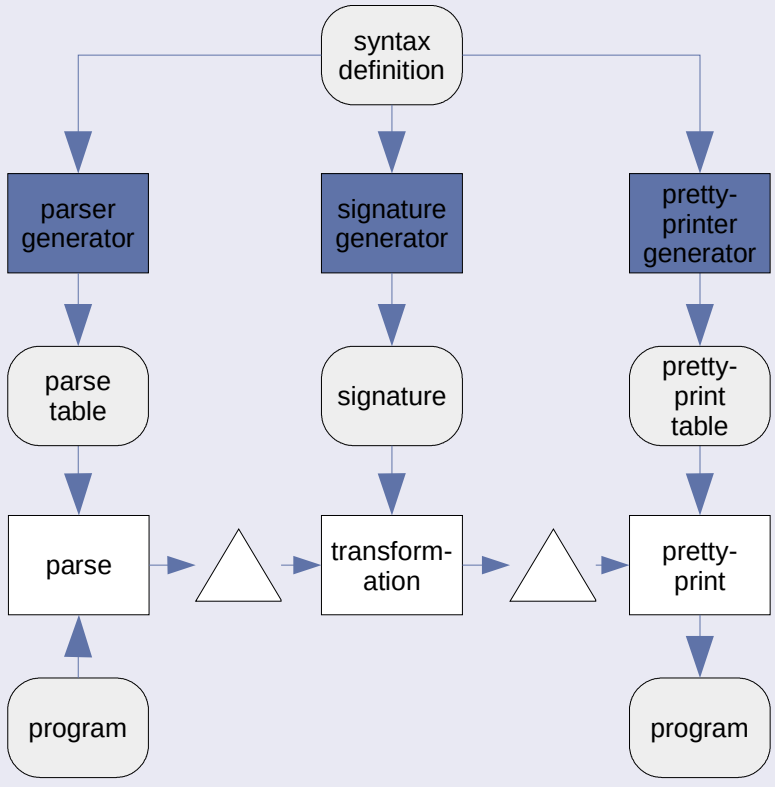
The Big Picture

A Simple Pipeline



1. Source code is *parsed* into a *structured* representation, a *concrete syntax tree* (CST).
2. The CST is pruned for non-essential information, to yield an *abstract syntax tree* (AST).
3. Additional information is added, e.g. type information.
4. One or multiple *transformations* are applied to the AST.
5. The result is serialized back to file.

The Small Picture



The *syntax definition* declares the *structure* of the language for the programs we transform and is used in constructing most pipeline stages.

Part II

Terms and Signatures

Terms

- Terms are a *structured representation* of programs.
- Their structure is often derived from the syntax definition of the language.
- The syntax definition gives rules for the structural validation of terms.

Terms are primarily used to encode *abstract syntax trees*.

Structural Definition of Annotated Terms

```
t  := bt                    -- basic term
    | bt { t }              -- annotated term

bt := C                     -- constant
    | C(t1,...,tn)          -- n-ary constructor
    | (t1,...,tn)           -- n-ary tuple
    | [t1,...,tn]           -- list
    | "ccc"                 -- quoted string
    | int                   -- integer
    | real                  -- floating point number
    | blob                  -- binary large object
```

Terms

Concrete Syntax

```
x := 1 + 2
```

Abstract Syntax Tree

```
Assign(x, Add(Int(1), Int(2)))
```

Why *abstract* syntax trees?

Concrete Syntax

$$x := 1 + 2$$

Accurate Concrete Syntax Tree


```
parsetree(appl(prod([cf(opt(layout)),cf(sort("Program")),cf(opt(layout))],sort("<START>"),no-at
trs),[appl(prod([],cf(opt(layout)),no-attrs),[]),appl(prod([cf(iter-star(sort("Stat")))],cf(sor
t("Program")),attrs([term(cons("Program"))])),[appl(list(cf(iter-star(sort("Stat")))),[appl(pro
d([lit("var"),cf(opt(layout)),cf(sort("Id")),cf(opt(layout)),lit(";")],cf(sort("Stat")),attrs([
term(cons("Declaration"))])),[lit("var"),appl(prod([cf(layout)],cf(opt(layout)),no-attrs),[32])
,appl(prod([lex(sort("Id"))],cf(sort("Id")),no-attrs),[appl(list(iter-star(char-class([range(0,
255)])))),[120])]),appl(prod([],cf(opt(layout)),no-attrs),[]),lit(";")]),appl(prod([cf(layout)],
cf(opt(layout)),no-attrs),[10]),appl(prod([cf(sort("Id")),cf(opt(layout)),lit(":="),cf(opt(layo
ut)),cf(sort("Exp")),cf(opt(layout)),lit(";")],cf(sort("Stat")),attrs([term(cons("Assign"))])),
[appl(prod([lex(sort("Id"))],cf(sort("Id")),no-attrs),[appl(list(iter-star(char-class([range(0,
255)])))),[120])]),appl(prod([cf(layout)],cf(opt(layout)),no-attrs),[32]),lit(":="),appl(prod([c
f(layout)],cf(opt(layout)),no-attrs),[32]),appl(prod([cf(sort("Exp")),cf(opt(layout)),lit("+"),
cf(opt(layout)),cf(sort("Exp"))],cf(sort("Exp")),attrs([term(cons("Add")),assoc(assoc)])),[appl
(prod([cf(sort("Int"))],cf(sort("Exp")),attrs([term(cons("Int"))])),[appl(prod([lex(sort("Int")
)],cf(sort("Int")),no-attrs),[appl(list(iter-star(char-class([range(0,255)]))),[49])])]),appl(p
rod([cf(layout)],cf(opt(layout)),no-attrs),[32]),lit("+"),appl(prod([cf(layout)],cf(opt(layout)
),no-attrs),[32]),appl(prod([cf(sort("Int"))],cf(sort("Exp")),attrs([term(cons("Int"))])),[appl
(prod([lex(sort("Int"))],cf(sort("Int")),no-attrs),[appl(list(iter-star(char-class([range(0,255
)])))),[50])])])]),appl(prod([],cf(opt(layout)),no-attrs),[]),lit(";")])])]),appl(prod([cf(layou
t)],cf(opt(layout)),no-attrs),[10])]),0)
```

Signatures

- In Stratego, signatures are *data declarations* that define the *structure* of terms.
- Terminology comes from the field of *universal algebra*; sorts with operations.
- A *signature* is a set of *constructors* on the form:
  - `name : sort * sort * ... -> sort`

Signatures

```
signature
 constructors
  Plus          : Exp * Exp -> Exp
  Assign        : Id * Exp -> Stat
  Int           : String -> Exp
  For           : Exp * Exp * Exp * List(Stat) -> Stat
  If            : Exp * List(Stat) * List(Stat) -> Stat
```

Part III

Transformation Rules

1. Program Transformation
2. Transformation Rules
3. Transformation Strategies
4. Dynamic Rules

Programs as Terms

Reminder

Programs can be represented as terms

```
sum k = 1 to n (a[i,k] * b[k,j])
```

```
Sum([Index("k",Int("1"),Var("n"))],
    Times(Subscript(Var("a"),[Var("i"),Var("k")]),
          Subscript(Var("b"),[Var("k"),Var("j")])))
```

Patterns

Term patterns can be used to analyze and compose programs

```
Sum([idx | idx*], e)
```

matches

```
Sum([Index("k",Int("1"),Var("n"))],
    Times(Subscript(Var("a"),[Var("i"),Var("k")]),
          Subscript(Var("b"),[Var("k"),Var("j")])))
```

Rewrite Rules

Definition

A rewrite rule `R : l -> r where c` replaces the pattern *l*, called the *left-hand side*, with *r*, the *right-hand side* when:

- *l* matches the *current term*
- *c* holds

Informally

Rewrite rules are basic units of tranformation, taking one term to another, based on structural pattern matching.

Rewrite Rules

```
SumSplit :
  Sum([idx | idx*], e) -> Sum([idx], Sum(idx*, e))
  where <not([])> idx*

DefSum :
  Sum([Index(x, e1, e2)], e3) ->
  Let([VarDec(y, NoTp(), Int("0"))],
      [For(Var(x), e1, e2,
           Assign(Var(y), Plus(Var(y), e3))),
       Var(y)])
  where new => y

LetFromAssign :
  Assign(lv, Let(d*, e*)) ->
  Let(d*, [Assign(lv, Seq([e*]))])
```

Term Rewriting

Term rewriting = normalization with respect to a set of rules

normalization = exhaustive application

normal form : no sub-term can be rewritten

Patterns in Concrete Syntax

- Problem: terms are hard to read, especially larger terms
- Observation: one-to-one correspondence between abstract syntax and concrete syntax
- Solution: use concrete syntax for patterns

```
SumSplit :
  |[ sum idx; idx* (e) ]| -> |[ sum idx(sum idx*(e)) ]|
  where <not([])> idx*

DefSum :
  |[ sum x = e1 to e2 ( e3 ) ]| ->
  |[ let var y := 0
      in for x := e1 to e2 do y := y + e3
         ; y
     end ]|
  where new => y
```

Systems that support Term Rewriting

- OBJ
- ASF+SDF
- Elan
- Maude
- TXL
- DMS
- FermaT
- ...

Part IV

Transformation Strategies

1. Program Transformation
2. Transformation Rules
3. Transformation Strategies
4. Dynamic Rules

Example: Compilation by Transformation

```
sum k = 1 to n
  (a[i,k] * b[k,j])
```

```
a_0 := 0;
k := 1;
d_0 := n;
t_3 := k <= d_0;
label f_0;
i_0 := not(t_3);
if i_0 goto g_0;
t_1 := a[i,k];
t_2 := b[k,j];
t_0 := t_1 * t_2;
a_0 := a_0 + t_0;
k := k + 1;
t_3 := k <= d_0;
goto f_0;
label g_0
```

```
a_0 := 0;
for k := 1 to n do
  (t_1 := a[i,k];
   t_2 := b[k,j];
   t_0 := t_1 * t_2;
   a_0 := a_0 + t_0)
```

```
a_0 := 0;
k := 1;
d_0 := n;
t_3 := k <= d_0;
while t_3 do
  (t_1 := a[i,k];
   t_2 := b[k,j];
   t_0 := t_1 * t_2;
   a_0 := a_0 + t_0;
   k := k + 1;
   t_3 := k <= d_0)
```

Controlling Application of Rules

Limitations of Term Rewriting

- Non-Termination
  - Exhaustive application may not terminate
- Non-Confluence
  - There may be diverging paths in rewrite relation
- Non-Selection
  - All rules are applied everywhere

Program transformation requires control over application of rules

Controlling Application of Rules

Common Solution: Functional Rewriting

Use extra constructors (called 'functions') to define control-point

- Traversal overhead: one rule for each constructor
- Tangling of rules and strategy: rules not reusable

```
Compile(e) ->
CollectDecls(SimpleExpressions(ForToWhile(Desugar(e))))

Desugar(|[ sum idx; idx+ (e) ]|) ->
Desugar(|[ sum idx(sum idx+ (e)) ]|)

Desugar(|[ sum x = e1 to e2 ( e3 ) ]|) ->
Desugar(|[ ... ]|

Desugar(|[ let d1* in e1* ]|) -> |[ let d2* in e2* ]|
where Desugar(d1*) => d2*; Desugar(e1*) => e2*
```

Staged Transformation

```
compile =
  for-with-while
  ; return-value
  ; mark-procedure-calls
  ; simple-expressions
  ; control-flow-to-goto
  ; collect-declarations
  ; use-return-register
  ; vars-on-stack
  ; add-stack-machine
  ; flatten-sequences
  ; unmark-procedure-calls
```

Compilation by sequence of transformations

```
for-with-while =
  topdown(try(ForToWhile))

simple-expressions =
  innermost(Simplify <+ LiftNonAtomicArgument <+ Desugar
            <+ LetSplit)
```

Controlling Application of Rules

Strategic Rewriting [Luttik & Visser 1997]

- Application of rules controlled by strategies
- Select rules and strategy
- Separation of rules and strategies
  - reuse of rules in multiple transformations
  - instantiate strategies with different rules
- Strategies are composed using strategy combinators
- Generic traversal reduces traversal overhead

Controlling Application of Rules

Strategic Rewriting [Luttik & Visser 1997]

- Application of rules controlled by strategies
- Select rules and strategy
- Separation of rules and strategies
  - reuse of rules in multiple transformations
  - instantiate strategies with different rules
- Strategies are composed using strategy combinators
- Generic traversal reduces traversal overhead

Strategic Programming [Laemmel, Visser & Visser 2003]

Language independent paradigm for programming with strategies, in particular, generic traversals.

Instantiated for rewriting, logic programming, functional programing, object-oriented programming

Sequential Composition

Sequential Composition

- Syntax: s_1 ; s_2
- Apply s_1, then s_2
- Fails if either s_1 or s_2 fails
- Variable bindings are propagated

```
Plus(Var("a"),Int("3"))
stratego> ?Plus(e1, e2); !Plus(e2, e1)
Plus(Int("3"),Var("a"))
```

Deterministic Choice

Deterministic Choice (Left Choice)

- Syntax: s_1 <+ s_2
- First apply s_1, if that fails apply s_2
- Note: local backtracking

```
PlusAssoc :
  Plus(Plus(e1, e2), e3) -> Plus(e1, Plus(e2, e3))
EvalPlus :
  Plus(Int(i),Int(j)) -> Int(k) where <addS>(i,j) => k
```

```
Plus(Int("14"),Int("3"))
stratego> PlusAssoc
command failed
stratego> PlusAssoc <+ EvalPlus
Int("17")
```

Identity and Failure

Identity

- Syntax: `id`
- Always succeed
- Some laws
  - `id ; s` $\equiv$ `s`
  - `s ; id` $\equiv$ `s`
  - `id <+ s` $\equiv$ `id`
  - `s <+ id` $\not\equiv$ `s`

Failure

- Syntax: `fail`
- Always fail
- Some laws
  - `fail <+ s` $\equiv$ `s`
  - `s <+ fail` $\equiv$ `s`
  - `fail ; s` $\equiv$ `fail`
  - `s ; fail` $\not\equiv$ `fail`

Defined Combinators

```
try(s)          = s <+ id

repeat(s)       = try(s; repeat(s))

while(c, s)     = if c then s; while(c,s) end

do-while(s, c) = s; if c then do-while(s, c) end
```

Traversal

Traversal Styles

Full traversal

- Folds: functional programming
- Visitors: object-oriented programming

Strategic programming

- One-level traversal combinator
  - descend to direct subterms
- Combine into multiple full traversal strategies
- Flavours
  - Congruence operator: specific for signature
  - Generic: independent of signature

Traversal Strategies

Visiting All Subterms

- Syntax: `all(`*s*`)`
- Apply strategy *s* to all direct sub-terms

```
Plus(Int("14"),Int("3"))
stratego> all(!Var("a"))
Plus(Var("a"),Var("a"))
```

Traversal Strategies

Visiting All Subterms

- Syntax: `all(`*s*`)`
- Apply strategy *s* to all direct sub-terms

```
Plus(Int("14"),Int("3"))
stratego> all(!Var("a"))
Plus(Var("a"),Var("a"))
```

```
bottomup(s)  = all(bottomup(s)); s
topdown(s)   = s; all(topdown(s))
downup(s)    = s; all(downup(s)); s
alltd(s)     = s <+ all(alltd(s))
innermost(s) = bottomup(try(s; innermost(s)))
```

Traversal Strategies

Visiting All Subterms

- Syntax: `all(s)`
- Apply strategy *s* to all direct sub-terms

```
Plus(Int("14"),Int("3"))
stratego> all(!Var("a"))
Plus(Var("a"),Var("a"))
```

```
bottomup(s)  = all(bottomup(s)); s
topdown(s)   = s; all(topdown(s))
downup(s)    = s; all(downup(s)); s
alltd(s)     = s <+ all(alltd(s))
innermost(s) = bottomup(try(s; innermost(s)))
```

```
for-with-while = topdown(try(ForToWhile))
simple-expressions =
  innermost(Simplify <+ LiftNonAtomicArgument <+ Desugar
            <+ LetSplit)
```

Traversal Strategies

Congruence Operator: Data-type Specific Traversal

- Syntax: $c(s_1, \ldots, s_n)$
  for each n-ary constructor c
- Apply strategies to direct sub-terms of a c term

```
Plus(Int("14"),Int("3"))
stratego> Plus(!Var("a"), id)
Plus(Var("a"),Int("3"))
```

```
mark-procedure-calls =
  rec mark(
    alltd(
      Assign(id, id)
      <+ While(id, mark)
      <+ For(id, id, id, mark)
      <+ If(id, mark, mark)
      <+ VarDec(id, id, id)
      <+ !Proc(<Call(id,id)>)))
```

Part V

Dynamic Rules

1. Program Transformation
2. Transformation Rules
3. Transformation Strategies
4. Dynamic Rules

Context-Sensitive Transformations

Problem: Rewrite Rules are Context-free

Rewrite rules can only access information in term that is matched

Context-Sensitive Transformations

Problem: 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

Context-Sensitive Transformations

Problem: 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

Solution: Dynamic Rewrite Rules

Define rewrite rules during transformation

Constant Folding

Constant folding

```
y := x * (3 + 4) ⇒ y := x * 7
```

Constant folding rules

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

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

AddZero : |[ 0 + e ]| -> |[ e ]|
```

Constant Folding

Constant folding

```
y := x * (3 + 4) ⇒ y := x * 7
```

Constant folding rules

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

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

AddZero : |[ 0 + e ]| -> |[ e ]|
```

Constant folding strategy (bottom-up)

```
EvalBinOp = EvalAdd <+ AddZero <+ EvalMul <+ EvalOther

try(s)    = s <+ id

constfold = all(constfold); try(EvalBinOp)
```

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)
```

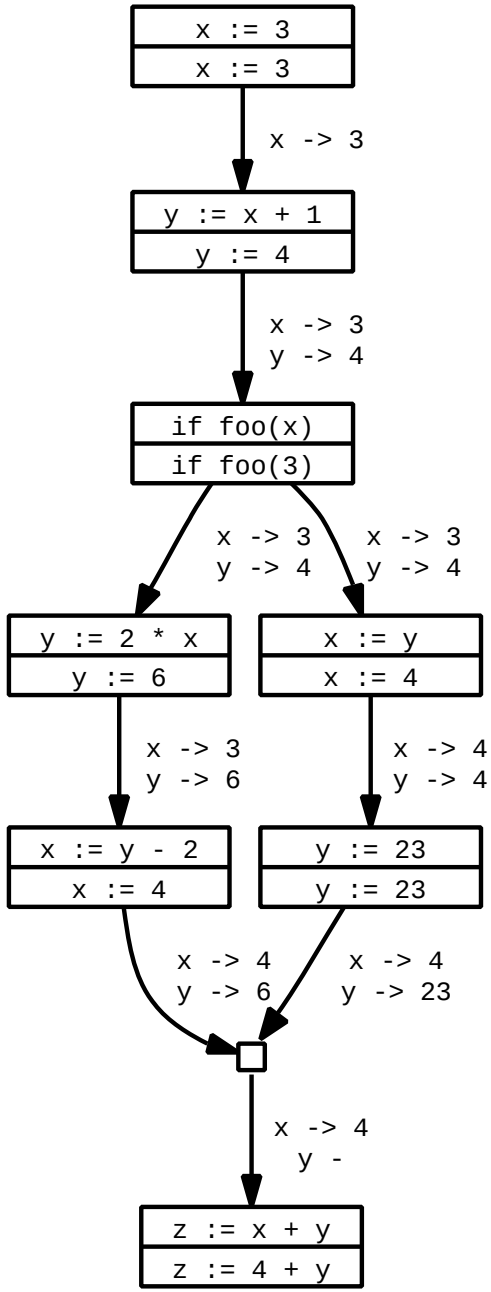
Flow-Sensitive Transformations

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



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))
```

$\Rightarrow$

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

Fixpoint Iteration

Flow-sensitive Constant Propagation

```
prop-const-while =
  ?⟦ while e1 do e2 ⟧
  ; (/PropConst\* ⟦while <prop-const> do <prop-const>⟧)
```

$$\texttt{/R* } s \equiv \texttt{((id /R\ } s\texttt{) /R\ } s\texttt{) /R\ ...)}$$

until fixedpoint of ruleset is reached

Fixpoint Iteration

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\ ...)`
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)
```

$\Rightarrow$

```
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)
```

$\Rightarrow$

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

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

Dependent Dynamic Rules

Dependent Dynamic Rules

Record *all* dependencies of dynamic rules in order to undefine all rules depending on a dependency
Use to define generic data-flow strategies

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

Tiny Tool Demo

Virtual Desktop Galore

Pay attention to Desktops 4 and 5.

Conclusion

- Stratego/XT 0.16 released last Friday
- Features:
  - Succinct, domain-specific languages
  - Collection of reusable components
  - Works *on* and *for* Linux, Unix, OSX, Windows
  - Licensed under the LGPL
  - Tutorial, examples and API documentation available online.
  - Editor plugin for Eclipse

`http://www.stratego-language.org`