Building Java Transformations with Stratego/XT

Martin Bravenboer [1], Karl Trygve Kalleberg [2], Eelco Visser[3]

http://www.stratego-language.org

[1,3] Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology, The Netherlands

[2] Department of Computer Science
University of Bergen, Norway

Tutorial OOPSLA/GPCE'06, October 23, Portland, Oregon

Part I

Introduction

Stratego/XT Project Mission Statement

Create a high-level, language parametric, rule-based program transformation system, which supports a wide range of transformations, admitting efficient implementations that scale to large programs.

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 Transformation Language and Tools

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

Java Transformation with Stratego/XT

- Instantiation of Stratego/XT to Java
- Language extension, DSL embedding, ...

Organization

Architecture and Infrastructure (45min)

- Parsing, pretty-printing, terms, typechecking, ...
- Martin Bravenboer

Local Transformations (45 min)

- Rewrite rules, strategies, traversal
- Eelco Visser

Break (30 min)

Type-Unifying Transformations (30 min)

- Collecting information
- Karl Trygve Kalleberg

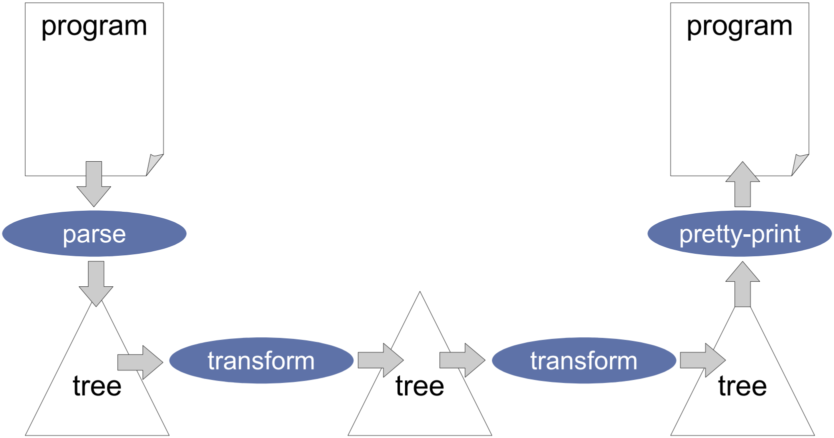
Context-Sensitive Transformations (60 min)

- Binding, dynamic rules, data-flow transformation
- Eelco Visser

Part II

Architecture & Infrastructure

Program Transformation Pipeline



Java Transformation Pipeline: Constant Propagation

```
public class Prop1 {
  public String getArg(String[] args) {
    int x = 3;
    int y = args.length;
    int z = 42;
    if(y > x) {
      z = 7 * x;
      x = x + 1;
    }
    else {
      x = x - 1;
      z = 19 + x;
    }
    y = x + z;
    return y;
  }
}
```

Java Transformation Pipeline: Constant Propagation

```
$ parse-java -i Prop1.java | ./java-propconst | pp-java
public class Prop1 {
  public String getArg(String[] args) {
    int x = 3;
    int y = args.length;
    int z = 42;
    if(y > 3) {
      z = 21;
      x = 4;
    }
    else {
      x = 2;
      z = 21;
    }
    y = x + 21;
    return y;
  }
}
```

Java Transformation Pipeline: Conditional Lifting

```java
public class Lift1
{
  public String getArg(String[] args)
  {
    return args.length > 0 ? args[0] : "";
  }
}
```

Java Transformation Pipeline: Conditional Lifting

```
$ dryad-front --tc on -i Lift1.java | \
    ./java-lift-conditional | core-lift-eblocks |  pp-java

public class Lift1
{
  public java.lang.String getArg(java.lang.String[] args)
  {
    java.lang.String expr_1;
    if(args.length > 0)
      expr_1 = args[0];
    else
      expr_1 = "";
    return expr_1;
  }
}
```

Architecture of Stratego/XT

Stratego

- Language for program transformation
- General purpose

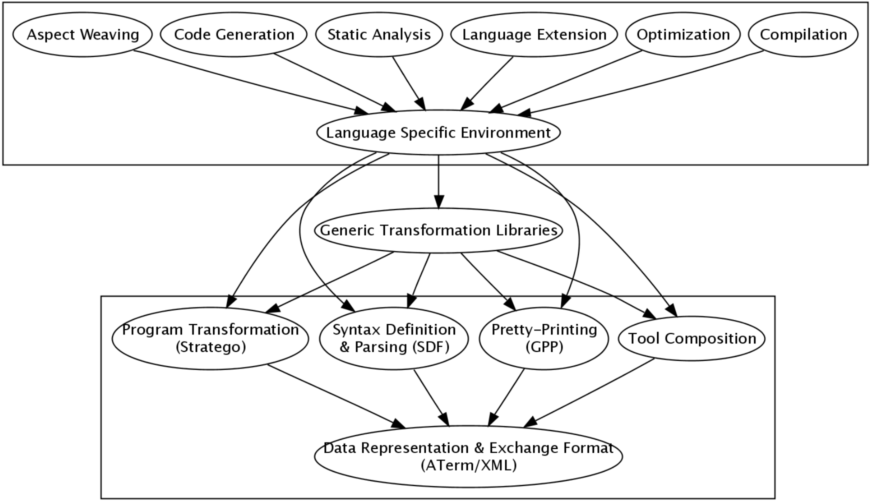
XT

- Collection of Transformation (X) Tools
- Infrastructure for implementing transformation systems
- Parsing, pretty-printing, program representation

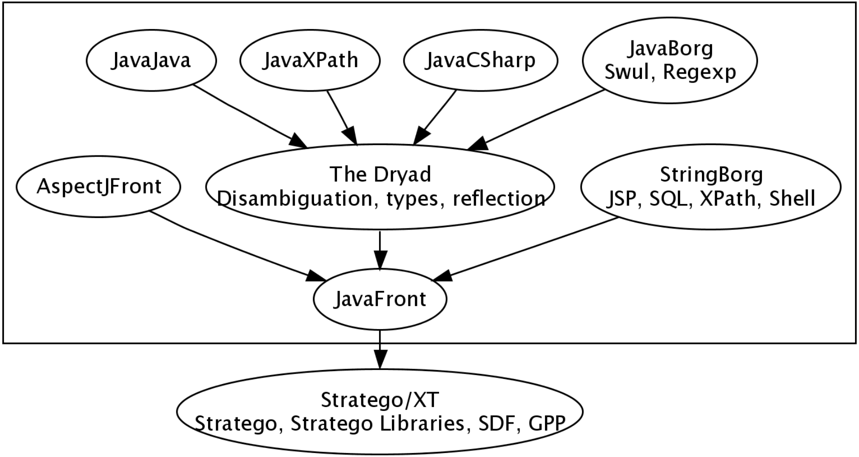
XT Orbit

- Instantiation for specific languages
- Java, JSP, AspectJ, BibTeX, C99, BibTeX, Prolog, PHP, SQL, XML, Shell, ECMAScript, . . .

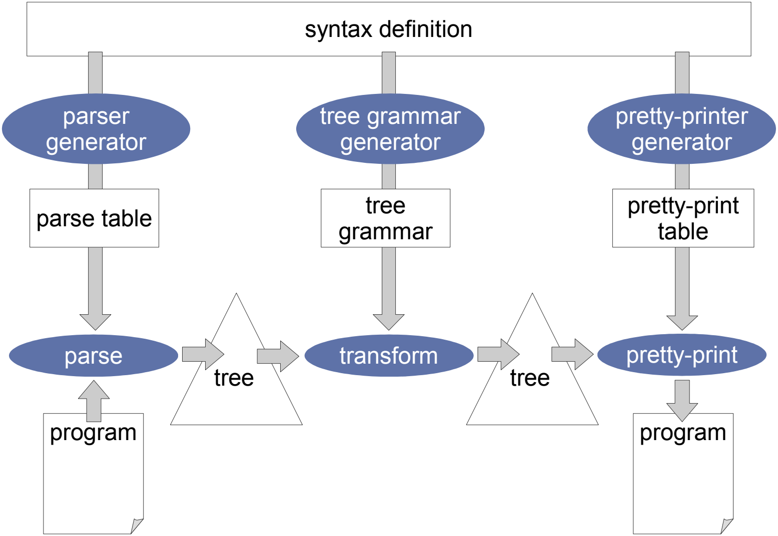
From Generic to Specific Components



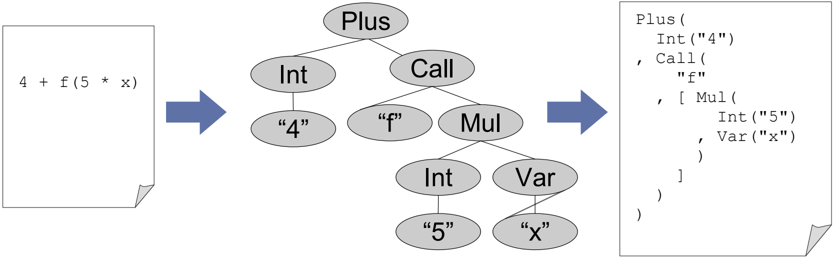
Transformation Infrastructure for Java



Architecture of Stratego/XT



Programs as Terms



Trees are represented as terms in the ATerm format

```
Plus(Int("4"), Call("f", [Mul(Int("5"), Var("x"))]))
```

ATerm Format

Application	`Void()`, `Call(`t`, `t`)`
List	`[]`, `[`t`, `t`, `t`]`
Tuple	`(`t`, `t`)`, `(`t`, `t`, `t`)`
Integer	`25`
Real	`38.87`
String	`"Hello world"`
Annotated term	t`{`t`, `t`, `t`}`

- Exchange of structured data
- Efficiency through maximal sharing
- Binary encoding

Structured Data: comparable to XML
Stratego: internal is external representation

Syntax Definition in Stratego/XT

SDF – Syntax Definition Formalism

1. **Declarative**
   - Important for code generation
   - Completely define the syntax of a language
2. **Modular**
   - Syntax definitions can be composed!
3. **Context-free and lexical syntax**
   - No separate specification of tokens for scanner
4. **Declarative disambiguation**
   - Priorities, associativity, follow restrictions
5. **All context-free grammars**
   - Beyond LALR, LR, LL

JavaFront

Syntax Definition: SDF grammar for Java 5

(i.e. generics, enums, annotations, . . .)

- Modular, structure of Java Specification, 3*rd* Edition
- Declarative disambiguation
  (i.e single expression non-terminal)
- Integrated lexical and context-free syntax
  Important for language extension (AspectJ)

Pretty Printer

- Modular, rewrite rules, extensible
- Preserves priorities (generated)
- Heavy testing: roundtrip

Parsing Java: CompilationUnit

```
$ echo "class Foo {}" | parse-java | pp-aterm
CompilationUnit(
  None()
, []
, [ ClassDec(
      ClassDecHead([], Id("Foo"), None(), None(), None())
    , ClassBody([])
    )
  ]
)
```

```
$ echo "package foo; class Foo" | parse-java | pp-aterm
CompilationUnit(
  Some(PackageDec([], PackageName([Id("foo")])))
, []
, [ ClassDec( ... ) ]
)
```

Parsing Java: Expressions and Types

```
$ echo "1 + x + xs[4]" | parse-java -s Expr | pp-aterm
Plus(
  Plus(Lit(Deci("1")), ExprName(Id("x")))
, ArrayAccess(ExprName(Id("xs")), Lit(Deci("4")))
)
```

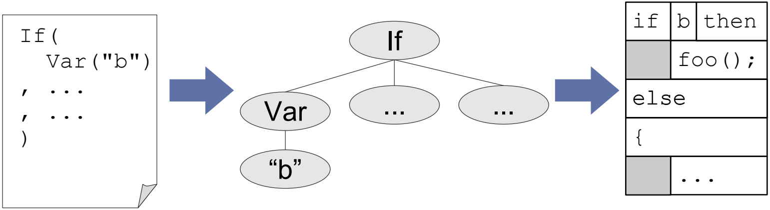
```
$ echo "this.y" | parse-java -s Expr | pp-aterm
Field(This(), Id("y"))
```

```
$ echo "x.y" | parse-java -s Expr | pp-aterm
ExprName(AmbName(Id("x")), Id("y"))
```

```
$ echo "String" | parse-java -s Type | pp-aterm
ClassOrInterfaceType(TypeName(Id("String")), None)
```

Pretty Printing in Stratego/XT

Code generators and source to source transformation systems need support for pretty printing.

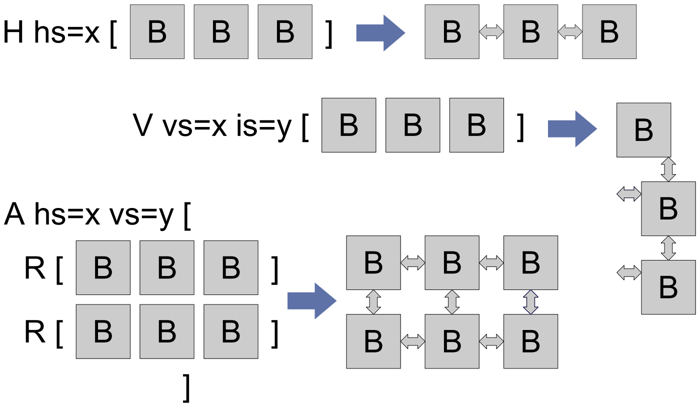


Stratego/XT: GPP (Generic Pretty Printing)

- Box language for text formatting
- Pretty printer generation or by hand
- Parenthesizer generation

Box Language

- Text formatting language
- Options for spacing, indenting
- 'CSS for plain text'



Other boxes: `HV`, `ALT`, `KW`, `VAR`, `NUM`, `C`

Example Box

```
V is=2 [
  H [KW["while"] "a" KW["do"]]
  V [
    V is=2 [
      H hs=1 [KW["if"] "b" KW["then"]]
      H hs=0 ["foo()" ";"]
    ]
    KW["else"]
    V [V is=2 ["{" "..."] "}"]
  ]
]
```

```
while a do
  if b then
    foo();
  else
  {
    ...
  }
```

Java Pretty Printer

```
$ cat Foo.java
public class Foo {
  public void bar() {
    if(true) {
      System.out.println("Stratego Rules!");
    }
  }
}

$ parse-java -i Foo.java | pp-java
public class Foo
{
  public void bar()
  {
    if(true)
    {
      System.out.println("Stratego Rules!");
    }
  }
}
```

Java Pretty Printer: Parentheses

```
Mul(
  Lit(Deci("1"))
, Plus(Lit(Deci("2")), Lit(Deci("3")))
)
$ pp-java -i Foo.jtree
1 * (2 + 3)
```

```
CastRef(
  ClassOrInterfaceType(TypeName(Id("Integer")), None())
, Minus(Lit(Deci("2")))
)
$ pp-java -i Foo.aterm
(Integer)(-2)
```

```
CastPrim(Int(), Minus(Lit(Deci("2"))))
$ pp-java -i Foo.aterm
(int)-2
```

Java Pretty Printer: Preserve Comments

```
public class Foo {
  /**
   * This method reports the universal truth.
   */
  public void bar() {
    // What an understatement!
    System.out.println("Stratego Rules!");
}}
$ parse-java --preserve-comments -i Foo.java | pp-java
public class Foo
{
  /**
   * This method reports the universal truth.
   */
  public void bar()
  {
    // What an understatement!
    System.out.println("Stratego Rules!");
  }
}
```

Architecture of Stratego/XT: Example

Collect SDF modules into a single syntax definition
```
$ pack-sdf -i Main.sdf -o TIL.def
```
Generate a parse-table
```
$ sdf2table -i TIL.def -o TIL.tbl
```
Parse an input file
```
$ sglri -i test1.til -p TIL.tbl
```
Generate pretty print table
```
$ ppgen -i TIL.def -o TIL.pp
```
Pretty print
```
$ sglri -i test1.til -p TIL.tbl | ast2text -p TIL.pp
$ sglri -i test1.til -p TIL.tbl | ast2text -p TIL-pretty.pp
```
Generate regular tree grammar and Stratego signature
```
$ sdf2rtg -i TIL.def -o TIL.rtg
$ rtg2sig -i TIL.rtg -o TIL.str
```
Generate and compile parenthesizer
```
$ sdf2parenthesize -i TIL.def -o til-parens.str
$ strc -i til-parens.str -m io-til-parens -la stratego-lib
```

Dryad, The Tree Nymph

Parsing Java often does not provide enough information for performing a program transformation.

- Ambiguous names and constructs
  - Type, package, or expression?
  - `java.awt.List` or `java.util.List`?
- Type information
  - Required for many transformations
- Basic definitions
  - Subtyping, conversions, method resolution, access control, ...
- Environment and program representation
  - Class hierarchies, unify source and bytecode
  - Access Java bytecode

Dryad R&Q: TypeName versus PackageName

```
import java.util.ArrayList;
```

Parse

```
TypeImportDec(
  TypeName(
    PackageOrTypeName(
      PackageOrTypeName(Id("java")), Id("util")
    )
  , Id("ArrayList")
  ))
```

Reclassify

```
TypeImportDec(
  TypeName(
    PackageName([Id("java"), Id("util")])
  , Id("ArrayList")
  ))
```

Dryad R&Q: AmbName

```
System.out.println("Hello World!");
```

Parse

```
MethodName(
  AmbName(AmbName(Id("System")), Id("out"))
, Id("println"))
```

Reclassify

```
MethodName(
  Field(
    TypeName(PackageName([Id("java"), Id("lang")])
    , Id("System"))
  , Id("out")
  )
, Id("println"))
```

Dryad Type Checker: Type Annotation

```
1 + 5
```

```
Plus(
  Lit(Deci("1")){ Type(Int) }
, Lit(Deci("5")){ Type(Int) }
){ Type(Int) }
```

```
"test " + 123
```

```
Plus(
  Lit(String([Chars("test")])) Type(String)
, Lit(Deci("123")){ Type(Int) }
){ Type(String) }
```

```
this
```

```
This{
  Type(ClassType(TypeName(PackageName([]), Id("Foo")), None))
}
```

Dryad Type Checker: Declaration Annotation

```
System.out.println("Hello World!")
```

```
Field(TypeName(java.lang.System), Id("out")) {
    Type(ClassType(java.io.PrintStream))
  , DeclaringClass(java.lang.System)
  }

Invoke(..., ...) {
    Type(Void)
  , CompileTimeDeclaration(
      MethodName(
        TypeName(java.io.PrintStream)
      , Id("println")
      , [ ClassType(java.lang.String) ]
      , Void
      )
    )
  }
```

Dryad Type Checker: Conversion Annotation

```
double d; d = 1;
```

```
Assign(...){ Type(Double), AssignmentConversion(
  [WideningPrimitiveConversion(Int, Double)]) }
```

```
Number n; n = 1;
```

```
Assign(...){ ..., AssignmentConversion(
    [ BoxingConversion(Int, RefInteger)
    , WideningReferenceConversion([RefNumber, RefInteger])
    ])}
```

```
List<String> list; list = new ArrayList();
```

```
Assign(...){ ..., AssignmentConversion(
    [ WideningReferenceConversion(
        [ Raw List
        , Raw AbstractList
        , Raw ArrayList
        ])
    , UncheckedConversion(Raw List, List<String>)
    ])}
```

Dryad Library

Dryad Model

- Representation of source and bytecode classes
- `repository` of available classes
- Classes, methods, fields, packages: lookup by name
- For example:
  - `get-superclass`, `get-inherited-methods`, `get-methods`, `get-fields` `get-declaring-class`, `get-formal-parameter-types`, ...

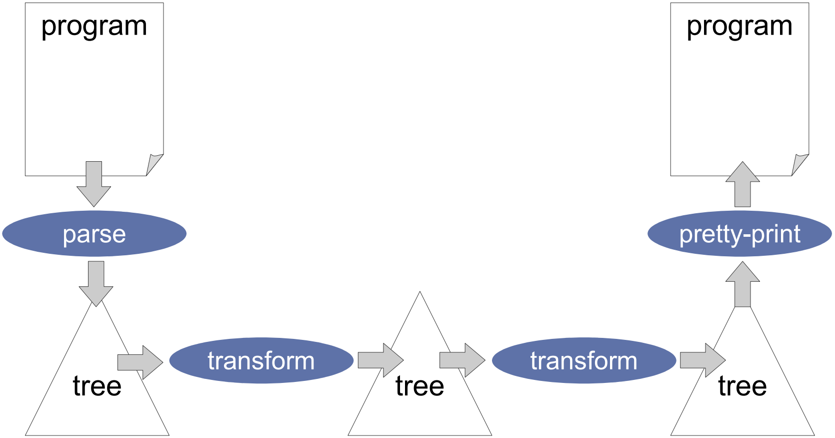
JLS definitions

- Conversions, types, access-control
- For example:
  - `is-subtype(|`*type*`)`
  - `is-assignment-convertable(|`*t*`)`,
  - `is-accessible-from(|`*from*`)`
  - `supertypes`

Part III

Realizing Program Transformations

How to Realize Program Transformations?



Implementing Transformation Components in Stratego

```
module trans

imports
  Java-15
  libstratego-lib

strategies

  main = io-wrap(...)

rules

  InvertIfNot :
    ... -> ...
```

Compile & Run

```
$ strc -i trans.str -la stratego-lib
$ parse-java -i MyClass.java |\
  trans |\
  pp-java
```

Interpret

```
$ parse-java -i MyClass.java |\
  stri -i trans.str |\
  pp-java
```

Interactive

```
$ parse-java -i MyClass.java |\
  stratego-shell
stratego> :show
CompilationUnit(None,[],[...])
```

Part IV

Rewrite Rules and Strategies

Term Rewriting

Conventional Term Rewriting

- Rewrite system = set of rewrite rules
- Redex = reducible expression
- Normalization = exhaustive application of rules to term
- (Stop when no more redices found)
- Strategy = algorithm used to search for redices
- Strategy given by engine

Strategic Term Rewriting

- Select rules to use in a specific transformation
- Select strategy to apply
- Define your own strategy if necessary
- Combine strategies

Transformation Strategies

A transformation strategy

- transforms the **current term** into a new term or **fails**
- may bind term variables
- may have side-effects (I/O, call other process)
- is composed from a few **basic operations and combinators**

Stratego Shell: An Interactive Interpreter for Stratego

```
<current term>
stratego> <strategy expression>
<transformed term>
stratego> <strategy expression>
command failed
```

Building and Matching Terms

Atomic actions of program transformation

1. Creating (building) terms from patterns
2. Matching terms against patterns

Build pattern

- Syntax: `!`*p*
- Replace current term by instantation of pattern *p*
- A pattern is a term with *meta-variables*

```
stratego> :binding e
e is bound to Var("b")
stratego> !Plus(Var("a"),e)
Plus(Var("a"),Var("b"))
```

Matching Terms

Match pattern

- Syntax: ?p
- Match current term (t) against pattern p
- Succeed if there is a substitution σ such that $\sigma(p) = t$
- Wildcard _ matches any term
- Binds variables in p in the environment
- Fails if pattern does not match

```
Plus(Var("a"),Int("3"))
stratego> ?Plus(e,_)
stratego> :binding e
e is bound to Var("a")
stratego> ?Plus(Int(x),e2)
command failed
```

Recognizing Dubious Statements and Expressions

if statement with empty branch; e.g. `if(x);`

```
?If(_, Empty(), _)
?If(_, _, Empty())
?If(_, Empty())
```

equality operator with literal `true` operand; e.g. `e == true`

```
?Eq(_, Lit(Bool(True())))
?Eq(Lit(Bool(True())), _)
```

Combining Match and Build

Basic transformations are combinations of match and build

Combination requires

1. Sequential composition of transformations
2. Restricting the scope of term variables

Syntactic abstractions (sugar) for typical combinations

1. Rewrite rules
2. Apply and match
3. Build and apply
4. Where
5. Conditional rewrite rules

Combining Match and Build

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"))
```

Combining Match and Build

Anonymous rewrite rule (sugar)

- Syntax: `(`p_1 `->` p_2`)`
- Match p_1, then build p_2
- Equivalent to: `?`p_1`;` `!`p_2

```
Plus(Var("a"),Int("3"))
stratego> (Plus(e1, e2) -> Plus(e2, e1))
Plus(Int("3"),Var("a"))
```

Combining Match and Build

Apply and match (sugar)

- Syntax: *s* `=>` *p*
- Apply *s*, then match *p*
- Equivalent to: *s*`;` `?`*p*

Build and apply (sugar)

- Syntax: `<`*s*`>` *p*
- Build *p*, then apply *s*
- Equivalent to: `!`*p*`;` *s*

```
stratego> <addS>("1","2") => x
"3"
stratego> :binding x
x is bound to "3"
```

Combining Match and Build

Term variable scope

- Syntax: `{`x_1`, . . . ,`x_n`:`s`}`
- Restrict scope of variables x_1,...,x_n to s

```
Plus(Var("a"),Int("3"))
stratego> (Plus(e1,e2) -> Plus(e2,e1))
Plus(Int("3"),Var("a"))
stratego> :binding e1
e1 is bound to Var("a")

stratego> {e3,e4:(Plus(e3,e4) -> Plus(e4,e3))}
Plus(Var("a"),Int("3"))
stratego> :binding e3
e3 is not bound to a term
```

Combining Match and Build

Where (sugar)

- Syntax: `where(`*s*`)`
- Test and compute variable bindings
- Equivalent to: `{x: ?x;` *s*`; !x}`
  for some fresh variable `x`

```
Plus(Int("14"),Int("3"))
stratego> where(?Plus(Int(i),Int(j)); <addS>(i,j) => k)
Plus(Int("14"),Int("3"))
stratego> :binding i
i is bound to "14"
stratego> :binding k
k is bound to "17"
```

Combining Match and Build

Conditional rewrite rules (sugar)

- Syntax: `(`$p1$ `->` p_2 `where` s`)`
- Rewrite rule with condition s
- Equivalent to: `(?`$p1$`; where(`s`); !`p_2`)`

```
Plus(Int("14"),Int("3"))
> (Plus(Int(i),Int(j)) -> Int(k) where <addS>(i,j) => k)
Int("17")
```

Naming and Composing Strategies

Reuse of transformation requires definitions

1. Naming strategy expressions
2. Named rewrite rules
3. Reusing rewrite rules through modules

Simple strategy definition and call

- Syntax: *f* = *s*
- Name strategy expression *s*
- Syntax: *f*
- Invoke (call) named strategy *f*

```
Plus(Var("a"),Int("3"))
stratego> SwapArgs = {e1,e2:(Plus(e1,e2) -> Plus(e2,e1))}
stratego> SwapArgs
Plus(Int("3"),Var("a"))
```

Named Rewrite Rules

Named rewrite rules (sugar)

- Syntax: f `:` $p1$ `->` p_2 `where` s
- Name rewrite rule $p1$ `->` p_2 `where` s
- Equivalent to: f `= {`x_1`,...,`x_n`: (`p_1 `->` p_2 `where` s`)}`
  (with x_1,...,x_n the variables in p_1, p_2, and s)

```
Plus(Var("a"),Int("3"))
stratego> SwapArgs : Plus(e1,e2) -> Plus(e2,e1)
stratego> SwapArgs
Plus(Int("3"),Var("a"))
```

Example: Inverting If Not Equal

```
if(x != y)
  doSomething();
else
  doSomethingElse();
```

$\Rightarrow$

```
if(x == y)
  doSomethingElse();
else
  doSomething();
```

```
InvertIfNot :
  If(NotEq(e1, e2), stm1, stm2) ->
  If(Eq(e1, e2), stm2, stm1)
```

Modules with Reusable Transformation Rules

```
module Simplification-Rules
rules
  PlusAssoc :
    Plus(Plus(e1, e2), e3) -> Plus(e1, Plus(e2, e3))

  EvalIf :
    If(Lit(Bool(True())), stm1, stm2) -> stm1

  EvalIf :
    If(Lit(Bool(False())), stm1, stm2) -> stm2

  IntroduceBraces :
    If(e, stm) -> If(e, Block([stm]))
    where <not(?Block(_))> stm
```

```
stratego> import Simplification-Rules
```

Composing Strategies

Rules define one-step transformations

Program transformations require many one-step transformations and selection of rules

1. Choice
2. Identity, Failure, and Negation
3. Parameterized and Recursive Definitions

Composing Strategies

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")
```

Composing Strategies

Guarded choice

- Syntax: s_1 `<` s_2 `+` s_3
- First apply s_1 if that succeeds apply s_2 to the result else apply s_3 to the original term
- Do not backtrack to s_3 if s_2 fails!

Motivation

- s_1 `<+` s_2 always backtracks to s_2 if s_1 fails
- `(`s_1`;` s_2`) <+ s3` $\not\equiv$ s_1 `<` s_2 `+` s_3
- commit to branch if test succeeds, even if that branch fails

```
    test1 < transf1
  + test2 < transf2
  + transf3
```

If then else (sugar)

- Syntax: `if` s_1 `then` s_2 `else` s_3 `end`
- Equivalent to: `where(`s_1`) <` s_2 `+` s_3

Composing Strategies

Identity

- Syntax: `id`
- Always succeed
- Some laws
  - `id ;` s $\equiv$ s
  - s `; id` $\equiv$ s
  - `id <+` s $\equiv$ `id`
  - s `<+ id` $\not\equiv$ s
  - s_1 `< id +` s_2 $\equiv$ s_1 `<+` s_2

Failure

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

Negation (sugar)

- Syntax: `not(`s`)`
- Fail if s succeeds, succeed if s fails
- Equivalent to: s `< fail + id`

Parameterizing Strategies

Parameterized and recursive definitions

- Syntax: $f(x_1,\ldots,x_n \mid y_1,\ldots,y_m)$ = s
- Strategy definition parameterized with strategies $(x_1,\ldots,x_n)$ and terms $(y_1,\ldots,y_m)$
- Note: definitions may be recursive

```
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
```

Part V

Traversal Strategies

1. In control of rewriting
   motivation for separation of rules and strategies
2. Programmable rewriting strategies
   some typical idioms for using traversal strategies
3. Realizing term traversal
   how traversal strategies are constructed

Term Rewriting for Program Transformation

Term Rewriting

- apply set of rewrite rules exhaustively

Advantages

- First-order terms describe abstract syntax
- Rewrite rules express basic transformation rules (operationalizations of the algebraic laws of the language.)
- Rules specified separately from strategy

Limitations

- Rewrite systems for programming languages often non-terminating and/or non-confluent
- In general: do not apply all rules at the same time or apply all rules under all circumstances

Term Rewriting for Program Transformation

```
signature
  sorts Prop
  constructors
    False : Prop
    True  : Prop
    Atom  : String -> Prop
    Not   : Prop -> Prop
    And   : Prop * Prop -> Prop
    Or    : Prop * Prop -> Prop
rules
  DAOL  : And(Or(x, y), z) -> Or(And(x, z), And(y, z))
  DAOR  : And(z, Or(x, y)) -> Or(And(z, x), And(z, y))
  DOAL  : Or(And(x, y), z) -> And(Or(x, z), Or(y, z))
  DOAR  : Or(z, And(x, y)) -> And(Or(z, x), Or(z, y))
  DN    : Not(Not(x))      -> x
  DMA   : Not(And(x, y))   -> Or(Not(x), Not(y))
  DMO   : Not(Or(x, y))    -> And(Not(x), Not(y))
```

This is a non-terminating rewrite system

Encoding Control with Recursive Rewrite Rules

Common solution

- Introduce additional constructors that achieve normalization under a restricted set of rules
- Replace a 'pure' rewrite rule

  p_1 `->` p_2

  with a functionalized rewrite rule:

  `f :` p_1 `->` p_2'

  applying f recursively in the right-hand side
- Normalize terms `f(t)` with respect to these rules
- The function now controls where rules are applied

Recursive Rewrite Rules

Map

```
map(s) : [] -> []
map(s) : [x | xs] -> [<s> x | <map(s)> xs]
```

Constant folding rules

```
Eval : Plus(Int(i), Int(j)) -> Int(<addS>(i,j))
Eval : Times(Int(i), Int(j)) -> Int(<mulS>(i,j))
```

Constant folding entire tree

```
fold : Int(i) -> Int(i)
fold : Var(x) -> Var(x)
fold : Plus(e1,e2) -> <try(Eval)>Plus(<fold>e1,<fold>e2)
fold : Times(e1,e2) -> <try(Eval)>Times(<fold>e1,<fold>e2)
```

Traversal and application of rules are tangled

Recursive Rewrite Rules: Disjunctive Normal Form

```
dnf   : True         -> True
dnf   : False        -> False
dnf   : Atom(x)      -> Atom(x)
dnf   : Not(x)       -> <not>(<dnf>x)
dnf   : And(x,y)     -> <and>(<dnf>x,<dnf>y)
dnf   : Or(x,y)      -> Or(<dnf>x,<dnf>y)

and1 : (Or(x,y),z) -> Or(<and>(x,z),<and>(y,z))
and2 : (z,Or(x,y)) -> Or(<and>(z,x),<and>(z,y))
and3 : (x,y)        -> And(x,y)
and  = and1 <+ and2 <+ and3

not1 : Not(x)       -> x
not2 : And(x,y)     -> Or(<not>(x),<not>(y))
not3 : Or(x,y)      -> <and>(<not>(x),<not>(y))
not4 : x            -> Not(x)
not  = not1 <+ not2 <+ not3 <+ not4
```

Analysis

Functional encoding has two main problems

Overhead due to explicit specification of *traversal*

- A traversal rule needs to be defined for each constructor in the signature and for each transformation.

Separation of rules and strategy is lost

- Rules and strategy are completely *intertwined*
- Intertwining makes it more difficult to *understand* the transformation
- Intertwining makes it impossible to *reuse* the rules in a different transformation.

Analysis

Language Complexity

Traversal overhead and reuse of rules is important, considering the complexity of real programming languages:

language	# constructors
Tiger	65
C	140
Java 5	325
COBOL	300–1200

Requirements

- Control over application of rules
- No traversal overhead
- Separation of rules and strategies

Programmable Rewriting Strategies

- Select rules to be applied in specific transformation
- Select strategy to control their application
- Define your own strategy if necessary
- Combine strategie

Idioms

- Cascading transformations
- One-pass traversal
- Staged transformation
- Local transformation

Strategic Idioms

Rules for rewriting proposition formulae

```
signature
  sorts Prop
  constructors
    False : Prop
    True  : Prop
    Atom  : String -> Prop
    Not   : Prop -> Prop
    And   : Prop * Prop -> Prop
    Or    : Prop * Prop -> Prop
rules
  DAOL  : And(Or(x, y), z) -> Or(And(x, z), And(y, z))
  DAOR  : And(z, Or(x, y)) -> Or(And(z, x), And(z, y))
  DOAL  : Or(And(x, y), z) -> And(Or(x, z), Or(y, z))
  DOAR  : Or(z, And(x, y)) -> And(Or(z, x), Or(z, y))
  DN    : Not(Not(x))      -> x
  DMA   : Not(And(x, y))   -> Or(Not(x), Not(y))
  DMO   : Not(Or(x, y))    -> And(Not(x), Not(y))
```

Strategic Idioms: Cascading Transformation

Cascading Transformations

- Apply small, independent transformations in combination
- Accumulative effect of small rewrites

```
simplify = innermost(R1 <+ ... <+ Rn)
```

disjunctive normal form

```
dnf = innermost(DAOL <+ DAOR <+ DN <+ DMA <+ DMO)
```

conjunctive normal form

```
cnf = innermost(DOAL <+ DOAR <+ DN <+ DMA <+ DMO)
```

Strategic Idioms: One-Pass Traversal

One-pass Traversal

- Apply rules in a single traversal over a program tree

```
simplify1  = downup(repeat(R1 <+ ... <+ Rn))
simplify2  = bottomup(repeat(R1 <+ ... <+ Rn))
```

constant folding

```
Eval : And(True, e) -> e
Eval : And(False, e) -> False
Eval : ...

eval = bottomup(try(Eval))
```

Strategic Idioms: One-Pass Traversal

Example: Desugarings

```
DefN  : Not(x)      -> Impl(x, False)
DefI  : Impl(x, y) -> Or(Not(x), y)
DefE  : Eq(x, y)   -> And(Impl(x, y), Impl(y, x))
DefO1 : Or(x, y)   -> Impl(Not(x), y)
DefO2 : Or(x, y)   -> Not(And(Not(x), Not(y)))
DefA1 : And(x, y)  -> Not(Or(Not(x), Not(y)))
DefA2 : And(x, y)  -> Not(Impl(x, Not(y)))
IDefI : Or(Not(x), y) -> Impl(x, y)
IDefE : And(Impl(x, y), Impl(y, x)) -> Eq(x, y)

desugar = topdown(try(DefI <+ DefE))

impl-nf  = topdown(repeat(DefN <+ DefA2 <+ DefO1 <+ DefE))
```

Strategic Idioms: Staged Transformation

Staged Transformation

- Transformations are not applied to a subject term all at once, but rather in stages
- In each stage, only rules from some particular subset of the entire set of available rules are applied.

```
simplify =
  innermost(A1 <+ ... <+ Ak)
  ; innermost(B1 <+ ... <+ Bl)
  ; ...
  ; innermost(C1 <+ ... <+ Cm)
```

Strategic Idioms: Local Transformation

Local transformation

- Apply rules only to selected parts of the subject program

```
transformation =
  alltd(
    trigger-transformation
    ; innermost(A1 <+ ... <+ An)
  )
```

Realizing Term Traversal

Requirements

- Control over application of rules
- No traversal overhead
- Separation of rules and strategies

Many ways to traverse a tree

- Bottom-up
- Top-down
- Innermost
- ...

What are the primitives of traversal?

Traversal Primitives

One-level traversal operators

- Apply a strategy to one or more direct subterms

Congruence: data-type specific traversal

- Apply a different strategy to each argument of a specific constructor

Generic traversal

- All: apply to all direct subterms
- One: apply to one direct subterm
- Some: apply to as many direct subterms as possible, and at least one

Congruence Operators

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"))
```

```
map(s) = [] + [s | map(s)])

fetch(s) = [s | id] <+ [id | fetch(s)]

filter(s) =
  [] + ([s | filter(s)] <+ ?[_|<id>]; filter(s))
```

Generic Traversal

Data-type specific traversal requires tedious enumeration of cases

Even if traversal behaviour is uniform

Generic traversal allows concise specification of default traversals

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))
```

```
const-fold =
  bottomup(try(EvalBinOp <+ EvalCall <+ EvalIf))
```

Generic Traversal: Desugaring

Example: Desugaring Expressions

```
DefAnd     : And(e1, e2) -> If(e1, e2, Int("0"))

DefPlus    : Plus(e1, e2) -> BinOp(PLUS(), e1, e2)

DesugarExp = DefAnd <+ DefPlus <+ ...

desugar    = topdown(try(DesugarExp))
```

```
IfThen(
  And(Var("a"),Var("b")),
  Plus(Var("c"),Int("3")))
stratego> desugar
IfThen(
  If(Var("a"),Var("b"),Int("0")),
  BinOp(PLUS,Var("c"),Int("3")))
```

Generic Traversal: Fixed-Point Traversal

Fixed-point traversal

```
innermost(s) = bottomup(try(s; innermost(s)))
```

Normalization

```
dnf = innermost(DAOL <+ DAOR <+ DN <+ DMA <+ DMO)
cnf = innermost(DOAL <+ DOAR <+ DN <+ DMA <+ DMO)
```

Visiting One Subterms

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

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

```
oncetd(s) = s <+ one(oncetd(s))
oncebu(s) = one(oncebu(s)) <+ s
spinetd(s) = s; try(one(spinetd(s)))
spinebu(s) = try(one(spinebu(s))); s
```

```
contains(|t) = oncetd(?t)
```

```
reduce(s) = repeat(rec x(one(x) + s))
outermost(s) = repeat(oncetd(s))
innermostI(s) = repeat(oncebu(s))
```

Generic Traversal: Some

Visiting some subterms (but at least one)

- Syntax: `some(s)`
- Apply strategy `s` to as many direct subterms as possible, and at least one

```
Plus(Int("14"),Int("3"))
stratego> some(?Int(_); !Var("a"))
Plus(Var("a"),Var("a"))
```

One-pass traversals

```
sometd(s) = s <+ some(sometd(s))
somebu(s) = some(somebu(s)) <+ s
```

Fixed-point traversal

```
reduce-par(s) = repeat(rec x(some(x) + s))
```

Summary

Summary

- Tangling of rules and strategy (traversal) considered harmful
- Separate traversal from rules
- One-level traversal primitives allow wide range of traversals

Part VI

Type-Unifying Transformations

Type Preserving vs Type Unifying

Transformations are type preserving

- Structural transformation
- Types stay the same
- Application: transformation
- Examples: simplification, optimization, ...

Collections are type unifying

- Terms of different types mapped onto one type
- Application: analysis
- Examples: free variables, uncaught exceptions, call-graph

Example Problems

`term-size`

- Count the number of nodes in a term

`occurrences`

- Count number of occurrences of a subterm in a term

`collect-vars`

- Collect all variables in expression

`free-vars`

- Collect all *free* variables in expression

`collect-uncaught-exceptions`

- Collect all *uncaught* exceptions in a method

List Implementation: Size (Number of Nodes)

Replacing `Nil` by `s1` and `Cons` by `s2`

```
foldr(s1, s2) =
  []; s1 <+ \ [y|ys] -> <s2>(y, <foldr(s1, s2)> ys) \
```

Add the elements of a list of integers

```
sum = foldr(!0, add)
```

Fold and apply `f` to the elements of the list

```
foldr(s1, s2, f) =
  []; s1 <+ \ [y|ys] -> <s2>(<f>y,<foldr(s1,s2,f)>ys) \
```

Length of a list

```
length = foldr(!0, add, !1)
```

List Implementation: Number of Occurrences

Number of occurrences in a list

```
list-occurrences(s) = foldr(!0, add, s < !1 + !0)
```

Number of local variables in a list

```
list-occurrences(?ExprName(id))
```

List Implementation: Collect Terms

Filter elements in a list for which s succeeds

```
filter(s) = [] + [s | filter(s)] <+ ?[_|<filter(s)>]
```

Collect local variables in a list

```
filter(ExprName(id))
```

Collect local variables in first list, exclude elements in second list

```
(filter(ExprName(id)),id); diff
```

Folding Expressions

Generalize folding of lists to arbitrary terms

Example: Java expressions

```
fold-exp(plus, minus, assign, cond, ...) =
 rec f(
    \ Plus(e1, e2) -> <plus>(<f>e1, <f>e2) \
  + \ Minus(e1, e2) -> <minus>(<f>e1, <f>e2) \
  + \ Assign(lhs, e) -> <assign>(<f>lhs, <f>e) \
  + \ Cond(e1, e2, e3) -> <cond>(<f>e1, <f>e2, <f>e3) \
  + ...
  )
```

Term-Size with Fold

```
term-size =
  fold-exp(MinusSize, PlusSize, AssignSize, ...)

MinusSize :
  Minus(e1 , e2) -> <add> (1, <add> (e1 , e2))

PlusSize :
  Plus(e1 , e2) -> <add> (1, <add> (e1 , e2))

AssignSize :
  Assign(lhs , e ) -> <add> (1, <add> (lhs , e))

// etc.
```

Limitations of Fold

Definition of fold

- One parameter for each constructor
- Define traversal for each constructor

Instantiation of fold

- One rule for each constructor
- Default behaviour not generically specified

Defining Fold with Generic Traversal

Fold is bottomup traversal:

```
fold-exp(s) =
  bottomup(s)

term-size =
  fold-exp(MinusSize <+ PlusSize <+ AssignSize <+ ...)
```

Definition of fold

- Recursive application to subterms defined generically
- One parameter: rules combined with choice

Instantiation: default behaviour not generically specified

Generic Term Deconstruction (1)

Specific definitions

```
MinusSize :
  Minus(e1 , e2) -> <add> (1, <add> (e1 , e2))

AssignSize :
  Assign(lhs , e ) -> <add> (1, <add> (lhs , e ))
```

Generic definition

```
CSize :
  C(e1 , e2 , ...) -> <add>(1,<add>(e1 ,<add>(e2 , ...)))
```

Requires generic decomposition of constructor application

Generic Term Deconstruction (2)

Generic Term Deconstruction

- Syntax: `?`p_1`#(`p_2`)`
- Semantics: when applied to a term $c(t_1, \ldots, t_n)$ matches
  - `"`c`"` against p_1
  - `[`$t_1, \ldots, t_n$`]` against p_2
- Decompose constructor application into its constructor name and list of direct subterms

```
Plus(Lit(Deci("1")), ExprName(Id("x")))
stratego> ?c#(xs)
stratego> :binding c
variable c bound to "Plus"
stratego> :binding xs
variable xs bound to [Lit(Deci("1")), ExprName(Id("x"))]
```

Definition of Crush

```
crush(nul, sum, s) :
  _#(xs) -> <foldr(nul, sum, s)> xs
```

Applications of Crush

```
node-size =
  crush(!0, add, !1)

term-size =
  crush(!1, add, term-size)

om-occurrences(s) =
  if s then !1 else crush(!0, add, om-occurrences(s)) end

occurrences(s) =
  <add> (<if s then !1 else !0 end>,
           <crush(!0, add, occurrences(s))>)
```

McCabe's cyclomatic complexity

```
public class Metric {
  public int foo() {
    if(1 > 2)
      return 0;
    else
      if(3 < 4)
        return 1;
      else
        return 2;
    if(5 > 6)
      return 3;
  }

  public int bar() {
    for(int i=0; i<5; i++) {}
  }
}
```

McCabe's cyclomatic complexity

- Computes the number of decision points in a function.
- Measure of minimum number of exection paths.
- Each control flow construct introduces another possible path.

```
cyclomatic-complexity =
  occurrences(is-control-flow)
  ; inc

is-control-flow =
  ?If(_, _)
  <+ ?If(_, _, _)
  <+ ?While(_, _)
  <+ ?For(_, _, _ ,_)
  <+ ?SwitchGroup(_, _)
```

NPATH complexity

```
public class Metric {
  public int foo() {
    if(1 > 2)
      return 0;
    else
      if(3 < 4)
        return 1;
      else
        return 2;
    if(5 > 6)
      return 3;
  }

  public int bar() {
    for(int i=0; i<5; i++) {}
  }
}
```

NPATH complexity

Complexity Analysis Algorithm (improved)

- Number of acyclic execution paths (not just nodes)
- Want to take into account the nesting of the control flow statements.
- Cost of a given control flow construct depends on its nesting level.

NPATH complexity: Implementation

```
npath-complexity =
  rec rec(
     ?Block(<map(rec)>)
     ; foldr(!1, mul)
  <+ {extra:
       is-control-flow
       ; where(extra := <AddPaths <+ !0>)
       ; crush(!0, add, rec)
       ; <add> (<id>, extra)
     }
  <+ is-BlockStm ; !1
  <+ crush(!0, add, rec)
  )

AddPaths: If(_, _) -> 1
AddPaths: While(_, _) -> 1
AddPaths: For(_, _, _, _) -> 1
```

Collect

Collect all (outermost) sub-terms for which s succeeds

```
collect(s) =
  ![<s>] <+ crush(![], union, collect(s))
```

Collect all sub-terms for which s succeeds

```
collect-all(s) =
  ![<s> | <crush(![], union, collect-all(s))>]
  <+ crush(![], union, collect-all(s))
```

Collect all local variables in an expression

```
get-exprnames = collect(ExprName(id))
```

Uncaught Exceptions (1)

Collect all uncaught exceptions

- Collect thrown exceptions
- Remove caught exceptions

Example

```
void thrower() throws
    IOException, Exception, NullPointerException { }

void g() throws Exception {
  try { thrower(); }
  catch(IOException e) {}
}
```

Uncaught exceptions: {NullPointerException, Exception}

Uncaught Exceptions (2)

Algorithm

- Recurse over the method definitions.
- Consider control constructs that deal with exceptions:
  - Method invocation and `throw` add uncaught exceptions.
  - `Try`/`catch` will remove uncaught exceptions.

```
collect-uncaught-exceptions =
  rec rec(
    ThrownExceptions(rec)
    <+ crush(![], union, rec)
  )
```

Uncaught Exceptions (3)

Handling `throw`

```
ThrownExceptions(rec):
  Throw(e) -> <union> ([<type-attr> e], children)
  where
    children := <rec> e
```

Handling method invocation

```
ThrownExceptions(rec):
  e@Invoke(o, args) -> <union> (this, children)
  where
    children := <rec> (o, args)
    ; <compile-time-declaration-attr> e
    ; lookup-method
    ; this := <get-declared-exception-types>
```

Uncaught Exceptions (4)

Handling `try/catch`

```
ThrownExceptions(rec):
  try@Try(body, catches) ->
            <union> (uncaught, <rec> catches)
  where
    uncaught := <rec; remove-all-caught(|try)> body
```

Summary

Summary

Generic term construction and deconstruction support the definition of generic analysis and generic translation problems

Next

Context-sensitive transformation problems

- bound variable renaming
- function/method inlining
- data-flow transformation
- interpretation

Solution: dynamic definition of rewrite rules

Part VII

Context-Sensitive Transformations

Context-Sensitive Transformation

Rewrite rules are context-free

- Rewrite rules
  - define local transformation of terms
  - no acces to context of terms transformed
- Strategies
  - control application of rules
  - not concerned with data

Many program transformations are context-sensitive

- Bound variable renaming
- Function inlining
- Data-flow transformations
- Partial evaluation
- Abstract interpretation

Binding

Need for most context-sensitive transformations arises from bindings

- Program are written as *text*
- Grammars overlay text with a *tree structure*
- Semantics refines trees to a *graph structure*
- Identifiers are placeholders for complex structures

Examples of binding types

- Modules
- Types
- Functions
- Variables

Binding: Modules

Module

- module name (in imports) refers to module definition
- `package foo.bar;`
- `import baz.*;`

Binding: Types

Type

- Type identifier refers to type definition
- Type definition: `class List { ... }`
- Variable declaration: `List x;`
- Casting: `(List) e`
- Inheritance: `class Stack extends List { ... }`

Functions and Methods

Function

- function call refers to function definition (body)
- definition: `int fib(int n) { ... }`
- call: `fib(y)`

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

Binding: Variables

Variable

- variable in expression refers to run-time value
- defined at an earlier stage in the program
- variable occurrence related to variable declaration and variable definition (assignment)

```
int x = e1;
...
x = x + 1;
print(x);
```

- there may be multiple possible definitions that affect a particular occurrence

```
x = e1;
if(cond) { x = e2; }
print(x);
```

Transformation: Constant Propagation

Replace variable occurrences by their values

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

redefinition of a variable

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

multiple bindings may reach same occurrence

Transformation: Common Subexpression Elimination

Replace expression with variable if computed before

```
x := a + b;
c := ...;
y := a + b;
```

expression rather than variable is 'bound'

Transformation: 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

Binding: Summary

Identifiers and bindings are fundamental in programming languages

Operations

Bound variable renaming

- replace variable declaration *and* all its uses by new name

Substitution

- replace occurrence of a variable by an 'expression'

Variable capture

- *without* accidentally binding a different variable

Evaluation = substitution + constant folding

- running a program requires replacing identifiers with their values and performing computations (folding)

How to extend rewriting to context-sensitive program transformation?

Part VIII

Dynamic Rules

Solution I: 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[Call(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
- no control over application of local rule

Implementation of Contextual Rules

```
UnfoldCall :
  Let(FunDef(f,[x],e1),e2) -> Let(FunDef(f,[x],e1),e3)
  where <alltd(
          {e4:(Call(f,e4) -> Let(VarDef(x,e4),e1))}
         )> e2 => e3
```

Observation: contextual rule performs local rewrite

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

Solution II: Dynamic Rewrite Rules

```
UnfoldCall :
  Let(FunDef(f,[x],e1),e2) -> Let(FunDef(f,[x],e1),e3)
  where <alltd(
          {e4:(Call(f,e4) -> Let(VarDef(x,e4),e1))}
         )> e2 => e3
```

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

Dynamic rules

- separate **definition** of contextual rule and its **application**
- 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

Part IX

Constant Propagation

Data-Flow Transformations

Propagation of (abstract) values from variable definitions to variable uses

- Constant propagation
- Copy propagation
- Common-subexpression elimination
- Partial evaluation

Propagation from uses to definitions

- Dead code elimination

Constant Propagation

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

⇒

```
(b := 1;
 c := 4;
 b := 2;
 b := z + 2;
 a := b + 4)
```

Ingredients of constant propagation

- constant folding (applying operations to constant values)
- propagation of constants from variable definitions to uses

Similar to evaluation, but

- produce 'residual' program if not all values known
- flow-sensitive: propagation may proceed differently in different branches

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 = b1 + 3;c
= 4;
b = foo();
a = b + c4
```

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

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

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

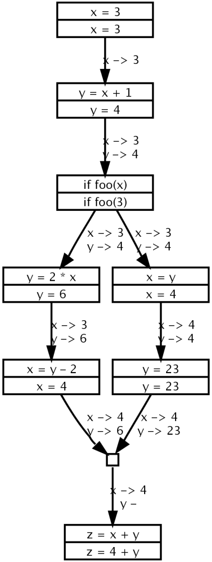
Flow-Sensitive Transformations

Flow-Sensitive Constant Propagation

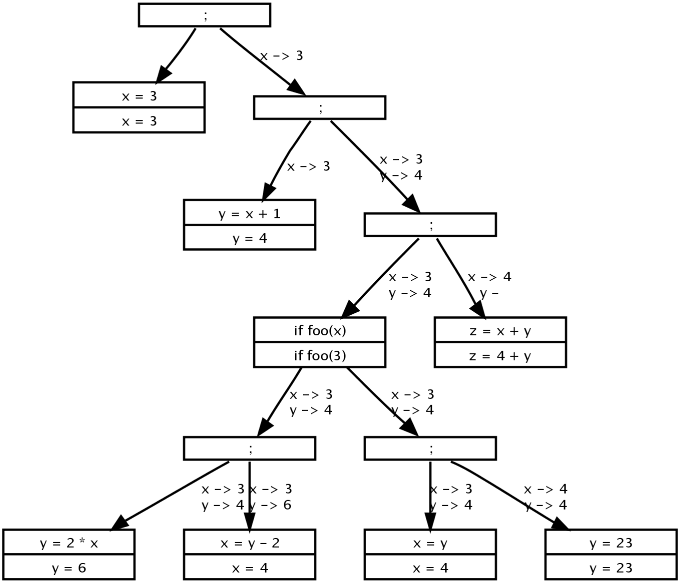
```
{x = 3;
 y = x + 1;
 if(foo(x))
   {y = 2 * x;
    x = y - 2;}
 else
   {x = y;
    y = 23;}
 z = x + y;}
```

```
{x = 3;
 y = 4;
 if(foo(3))
   {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>) <id> else <id> ]|
  ; (|[if(<id>) <prop-const> else <id>]|
       /PropConst\ |[if(<id>) <id> else <prop-const>]|)
```

s_1 `/R\` s_2: fork and intersect

Propagation through Loops

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

$\Rightarrow$

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

Fixed-Point Intersection of Rule Sets

```
{ int w = 20, x = 20,
      y = 20, z = 10;
  while(SomethingUnknown()) {
    if x = 20 then w = 20 else w = 10;
    if y = 20 then x = 20 else x = 10;
    if z = 20 then y = 20 else y = 10;}
   w; x; y; z; }
```

```
{ int w = 20, x = 20,
      y = 20, z = 10;
  while(SomethingUnknown()) {
    if x = 20 then w = 20 else w = 10;
    if y = 20 then x = 20 else x = 10;
    y = 10;}
   w; x; y; 10; }
```

	w	x	y	z
	20	20	20	10
1	20	20	10	10
	20	20	-	10
2	20	-	10	10
	20	-	-	10
3	-	-	10	10
	-	-	-	10
4	-	-	10	10
	-	-	-	10

Fixpoint Iteration

Flow-sensitive Constant Propagation

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

`/R\* s ≡ ((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)
  i = 3;
else
  z = foo();
print(i);
```

⇒

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

```
EvalIf : |[ if(false) e1 else e2 ]| -> |[ e2 ]|
EvalIf : |[ if(true) e1 else e2 ]| -> |[ e1 ]|
```

```
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) {
   if(x == 10)
     dosomething();
   else {
     dosomethingelse();
     x = x + 1;
   }
 y = x;}
```

⇒

```
{x = 10;
 while(A)
   dosomething();
 y = 10;}
```

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

Local Variables

```
{ int x = 17;
  { int y = x + 1;
     { int x = y+1;
        ... }
  }
  print(x);
}
```

$\Rightarrow$

```
{ int x = 17;
  { int y = 18;
     { int x = 19;
        ... }
  }
  print(17);
}
```

propagation rules should only be applied when the subject variable is in scope

Scope Labels – Constant Propagation with Local Variables

```
{ int a = 1, b = 2, c = 3;
  a = b + c;
  { int c = a + 1;
    b = b + c;
    a = a + b;
    b = z + b;}
  a = c + b + a; }
```

⇓

```
{ int a = 1, b = 2, c = 3;
  a = 5;
  { int c = 6;
    b = 8;
    a = 13;
    b = z + 8;}
  a = 3 + b + 13; }
```

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

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

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

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

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

can this be applied to other data-flow transformations?

Stratego/XT Transformation Language and Tools

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

Java Transformation with Stratego/XT

- Instantiation of Stratego/XT to Java
- Language extension, DSL embedding, ...

http://www.stratego-language.org

The End