Model-to-Model Transformation by Term Rewriting

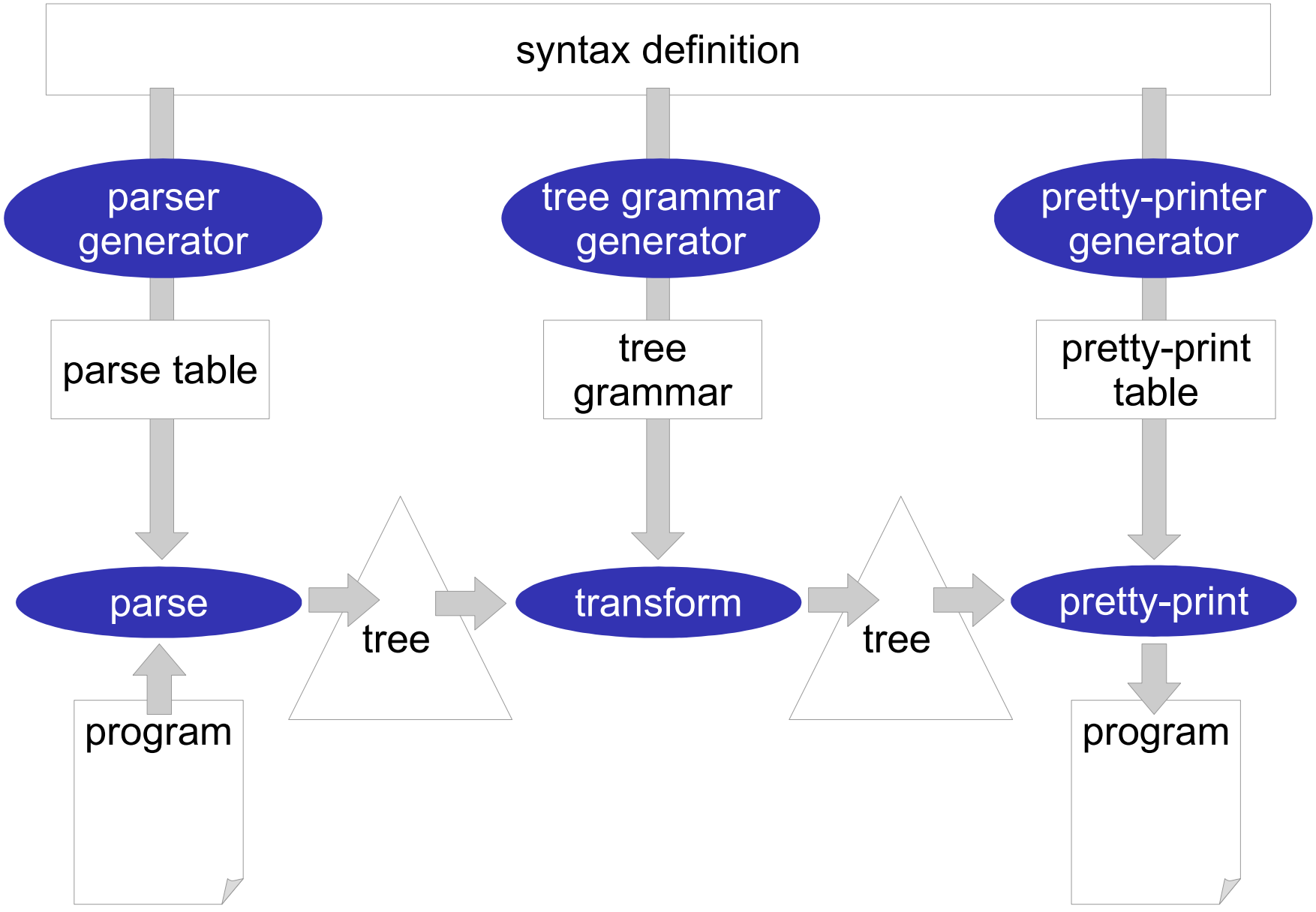
Eelco Visser

Delft University of Technology

http://www.strategoxt.org

Code Generation 2009
Cambridge
June 16, 2009

Transformation Infrastructure



Realizing Program Transformations

How to realize transformations of (abstract syntax) trees?

Stratego

Stratego is a language for program transformation based on term rewriting with programmable rewriting strategies.

Terms

The Annotated Term 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}

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

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 current term into a new term or fails
- may bind term variables
- may have side-effects (I/O, call other process)

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,[],[...])
```

Anatomy of a Stratego Program (1)

pretty-print.str

```
module pretty-print

// parse a NanoWebDSL file and pretty-print it

imports
  libstratego-lib
  libstratego-xtc
  libwebdsl-front
  remove-annos
```

Anatomy of a Stratego Program (2)

pretty-print.str

```
strategies

  main =
    xtc-io-wrap(
        webdslc-options
      , webdslc-usage
      , webdslc-about
      , ![] // xtc dependencies
      , xtc-pretty-print
    )

  xtc-pretty-print = id
    ; set-appname
    ; set-default-config
    ; xtc-parse-webdsl
    ; remove-position-annos
    ; output-webdsl
```

Exercise 3.1: Pretty-Printing

Using a library

- `cd strategoxt-tutorial/rewriting`
- inspect: `pretty-print.str`
- inspect: `Makefile`
- build: `make pretty-print`
- apply: `./pretty-print -i test/test1.app`
- try: `./pretty-print --help`
- note: we are using parser and pretty-printer from `libwebdsl-front`

Normalizing WebDSL Models

Source: test/testcall.app

```
application tasks
define page home() {
  section("Users") {
    "content"
  }
}
```

Target: test/testcall-norm.app

```
application tasks
define page home () {
  section(){
    header(){
      output("Users"){}
    }
    text("content"){}
  }
}
```

Setup

normalize-ast.str

```
module normalize
  // parses a NanoWebDSL file,
  // applies syntactic normalizations to the AST,
  // and pretty-prints the result
imports ... // same as pretty-print.str

strategies

  main = xtc-io-wrap(..., xtc-normalize)

  xtc-normalize = id
    ; set-appname
    ; set-default-config
    ; xtc-parse-webdsl
    ; normalize           // the strategy we need to write
    ; remove-position-annos
    ; output-webdsl
```

Transforming Terms

Source

```
$ ./pretty-print -i test/testcall.app --no-pp | pp-aterm
Application("tasks", [
  Define([Page()], "home", [], [
    TemplateCall("section",
      [String("\"Users\"")],
      [Text("\"content\"")])])])
```

Target

```
$ ./pretty-print -i test/testcall-norm.app --no-pp
Application("tasks", [
  Define([Page()], "home", [], [
    TemplateCall("section", [], [
      TemplateCall("header", [], [
        TemplateCall("output",[String("\"Users\"")],[])]),
      TemplateCall("text",[String("\"content\"")],[])])])])
```

Rewrite Rules

Rewrite Rule

```
L : p1 -> p2
```

Example

```
Assoc :
  Plus(Plus(e1, e2), e3) -> Plus(e1, Plus(e2, e3))
```

Deriving Transformations

Source

```
Text("\"content\"")
```

Target

```
TemplateCall("text",[String("\"content\"")],[])
```

Rewrite Rule

```
normalize-text :
  Text(str) -> TemplateCall("text", [String(str)], [])
```

Deriving Transformations

Source

```
TemplateCall("section",[...],[...])
```

Target

```
TemplateCall("section", [], [
  TemplateCall("header", [], [...])
  ...
])
```

Rewrite Rule

```
normalize-section :
  TemplateCall("section", [e], elem*) ->
  TemplateCall("section", [],
    [TemplateCall("header", [],
       [TemplateCall("output", [e], [])])
    |elem*])
```

Combining Rules with Strategies

```
strategies

  innermost-rep(s) =
    all-consnil(innermost-rep(s))
    ; try(s ; innermost-rep(s))

  all-consnil(s) =
    ?[_|_] < [s|s] + all(s)

strategies
  normalize =
    innermost-rep(desugar)

  desugar =
    normalize-text <+ normalize-section
```

Exercise 3.2: Rewrite Rules

Normalizing with rewrite rules

Inspect

- `normalize-ast.str`

Build

- `make normalize-ast`

Test

- `./normalize-ast -i test/test2.app`

Extend

- `normalize-ast.str`

Goal

- `./normalize-ast -i test/test2.app == test/test2-norm.app`

Exercise 3.2: Answer

See `normalize-ast-32.str`

Conditional Rewrite Rules

Strategy Parameters

```
map(f) : [] -> []

map(f) : [x | x*] -> [y | y*]
  where y  := <f> x
      ; y* := <map(f)> x*
```

Term Parameters

```
inverse = inverse(|[])

inverse(|y*) : [] -> y*

inverse(|y*) : [x | x*] -> y*
  where y* := <inverse(|[x | y*])> x*
```

Exercise 3.3: Conditional Rewrite Rules (1)

Source: test/testcase.app

```
define page task(task : Task, tab : String) {
  case(tab) {
    "view" { viewTask(task) }
    "edit" { editTask(task) }
  }
}
```

Target: test/testcase-norm.app

```
define page task (task : Task, tab : String) {
  var caseval0 : String := tab ;
  if (caseval0 == "view") { viewTask(task){} }
  else {
    if (caseval0 == "edit") { editTask(task){} }
    else { }
  }
}
```

Exercise 3.3: Conditional Rewrite Rules (2)

Writing rewrite rules: Case Statement

rewrite case statement in terms of if-then-else

- extend `normalize-ast.str`

Inspect

- `./pretty-print -i test/testcase.app --no-pp`
- `./pretty-print -i test/testcase-norm.app --no-pp`

Tips

- `<newname> string` produces a unique new name
- use `dummy{ ... }` to wrap list of elements

Exercise 3.3: Answer (normalize-ast-case.str)

```
normalize-ui :
  TemplateCase(e, talt*) ->
  TemplateCall("dummy",[],[VarDeclInit(x, srt, e)|[elem]])
  where srt := SimpleSort("String")
        ; x := <newname> "caseval"
        ; elem := <template-case-to-if(|x)> talt*

template-case-to-if(|x):
  [] -> TemplateCall("dummy", [], [])

template-case-to-if(|x):
  [TemplateCaseAlt(const, elem1*), talt*] ->
  IfTempl(Eq(Var(x), const), elem1*, [elem2])
  where elem2 := <template-case-to-if(|x)> talt*

template-case-to-if(|x):
  [TemplateCaseAltDefault(elem1*), talt*] ->
  TemplateCall("dummy", [], elem1*)
```

Rewriting with Concrete Object Syntax

Abstract Syntax

```
normalize-section :
  TemplateCall("section", [e], elem*) ->
  TemplateCall("section", [],
    [TemplateCall("header", [],
       [TemplateCall("output", [e], [])])
    |elem*])
```

Concrete Syntax

```
normalize-section :
  elem|[ section(e){ elem* } ]| ->
  elem|[ section(){ header{output(e)} elem* } ]|
```

Combining Meta and Object Language

syntax/WebDslMix.sdf (fragment)

```
module WebDslMix[E]
imports WebDSL
exports
  context-free syntax
    "elem" "|[" TemplateElement  "]|" -> E {cons("ToMetaExpr")}
    "elem*""|[" TemplateElement* "]|" -> E {cons("ToMetaExpr")}
  variables
    "elem"[0-9]*    -> TemplateElement  {prefer}
    "elem"[0-9]* "* -> TemplateElement* {prefer}
```

syntax/StrategoWebDSL.sdf

```
module StrategoWebDSL
imports
  StrategoMix[StrategoHost]
  WebDslMix[ Term[[StrategoHost]] ]
hiddens
  context-free start-symbols Module[[StrategoHost]]
```

Inspecting Underlying Term Structure

normalize.meta

```
Meta([Syntax("StrategoWebDSL")])
```

```
$ pp-stratego -i normalize.str -I ../syntax/ | less
...
  normalize-ui :
    TemplateCall("section", [e], elem*) ->
    TemplateCall("section", [], [
      TemplateCallBody("header", [
        TemplateCallNoBody("output", [e])])
        | elem*])
...
```

Exercise 3.4: Concrete Syntax

Case statement normalization with concrete syntax

Extend

- `normalize.str`

Tips

- `talt*|[ ... ]|` wraps list of template case alternatives
- `talt*` is meta-variable for such a list
- `const` is meta-variable for constant values
- example: `talt*|[ const{ elem* } talt* ]|`

Exercise 3.4: Answer (normalize-case.str)

```
normalize-ui :
  elem|[ case(e) { talt* } ]| ->
  elem|[ dummy(){ var x : srt := e; elem } ]|
  where srt  := SimpleSort("String")
      ; x    := <newname> "caseval"
      ; elem := <template-case-to-if(|x)> talt*

template-case-to-if(|x) :
  talt*|[ ]| -> elem|[ dummy(){ } ]|

template-case-to-if(|x) :
  talt*|[ const{ elem1* } talt* ]| ->
  elem|[ if(x == const) { elem1* } else { elem2 } ]|
  where elem2 := <template-case-to-if(|x)> talt*

template-case-to-if(|x) :
  talt*|[ default { elem1* } talt* ]| ->
  elem|[ dummy(){ elem1* } ]|
```

Context-Sensitive Transformations

Source

```
entity Task {
  description :: String (name)
  done        :: Bool
}
define page task(task : Task) {
  derive editPage from task
}
```

Target

```
define page task (task : Task) {
  header(){ text("Edit "){} output(task.name){} }
  form(){
    group("Details"){
      groupitem(){ label(Description: ){ input(task.description){} } }
      groupitem(){ label(Done: ){ input(task.done){} } }
    }
    group(){ action(Save, save()){} }
  }
  action save(){ task.save(); return task(task); }
}
```

The Derive editPage Transformation (1)

```
desugar-derive :
  elem|[ derive editPage from e ]| ->
  elem|[ derive editPage from e for (dprop*) ]|
  with SimpleSort(t) := <type-of> e
     ; prop*  := <Properties> t
     ; dprop* := <filter(property-to-derive-prop)> prop*
```

The Derive editPage Transformation (2)

```
derive-page :
  elem|[ derive editPage from e for (dprop*) ]| ->
  elem|[
    dummy() {
      header{"Edit " text(e.name) }
      form {
        group("Details") {
          derive editRows from e for (dprop*)
        }
        group() {
          action("Save", save())
        }
      }
      action save() {
        e.save();
        return x_view(e);
      }
    }
  ]|
  with  SimpleSort(srt) := <type-of> e
      ; x_view := <decapitalize-string> srt
```

Dynamic Rewrite Rules (1)

```
desugar-derive :
  ...
     ; prop* := <Properties> t

declare-all =
  alltd(declare)

declare =
  ?|[ entity x_class { prop* } ]|
  ; rules(
      Properties : x_class -> prop*
    )
```

Dynamic Rewrite Rules (2)

```
desugar-derive : ...
  with SimpleSort(t) := <type-of> e

type-of :
  Var(x) -> srt
  where srt := <TypeOf> x

rename-bound(|srt) :
  x -> y
  with y := x{<newname> x}
     ; rules (
         Rename : Var(x) -> Var(y)
         TypeOf : y -> srt
       )
```

Dynamic Rewrite Rules (3)

```
rename-all = alltd(rename)

rename = Rename

rename :
  Arg(x, srt) -> Arg(y, srt)
  with y := <rename-bound(|srt)> x

rename :
  def |[ define mod* x(farg1*) { elem1* } ]| ->
  def |[ define mod* x(farg2*) { elem2* } ]|
  with {| Rename
        : farg2* := <map(rename)> farg1*
        ; elem2* := <rename-all> elem1*
        |}
```

Exercise 3.5: Dynamic Rules

New binding construct

```
define page tasks(user : User) {
  for(task : Task in user.tasks) {
    derive editPage from task
  }
}
```

Extend rename rules to deal with iteration