Bidirectional Attribute Grammars and their use in Extensible Languages

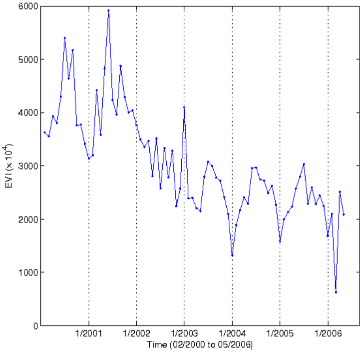
Eric Van Wyk[1] & João Saraiva

University of Minnesota & University of Minho

WG 2.11

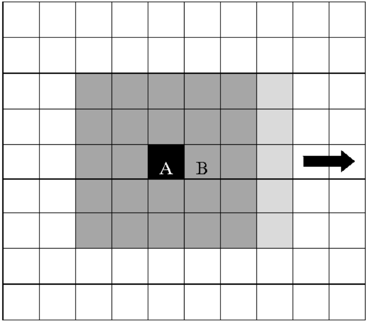
March 3, 2010, St. Andrews

[1] This work is partially funded by NSF grant #0905581



Source: Google Maps.

- scale matters
- interest in MapReduce style computations
- FP language extensions to C



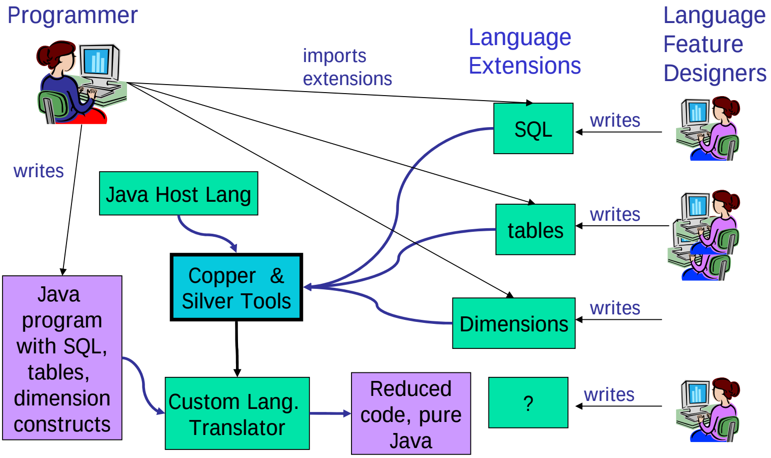
- *e.g.* variations on `map`
  - 2D, 3D, time series
  - context dependent
  - missing, erroneous data
- optimizations, code generation

Extensible Languages

- Extensible language frameworks allow
  - language designers to build language extensions (plug-ins) that define
    - new syntax
    - new semantics
    - new optimizations
  - and programmers to create a customized language from a chosen set of extensions.

    Tools for composition and translator generation.
- Silver - an extensible AG system
- Copper - a parser and context-aware scanner generator

  Verifiable composition of grammar or parse table fragments


Programmer
imports
extensions
Language
Extensions
Language
Feature
Designers
writes
SQL
writes
Java Host Lang
tables
writes
Copper &
Silver Tools
Dimensions
writes
Java
program
with SQL,
tables,
dimension
constructs
Custom Lang.
Translator
Reduced
code, pure
Java
?
writes

Data intensive computing project

- extensions to C, using Silver and Copper
- reasonable syntactic extensions
- optimization at the high level
- generation of efficient (parallel) C code
- recognizable generated C code

bidirectional transformations

- source: concrete syntax of extended C
- target: abstract syntax of extended C
- modifications:
  - optimization of extended C
  - compilation of extensions to plain C
- *get*: CST to AST
- mod: optimization and translation of extensions to plain C, on AST
- *put*: AST to CST
- Note: editing the generated code is not the goal.

bidirectional transformations (2)

- challenge - source is richer than the target
- access to the original source is used to help in computing the *put* function back after transformations on the target.
- here, more productions in concrete than abstract syntax
- also, maintaining white space and comments
  - though not part of this talk

BX in Attribute grammars

Attribute grammars with

- reference attributes,
- forwarding, and
- a modified notion of pattern matching,

provide an nice means for implementing bidirectional transformations.

Extensibility lets us add extensions to generate the *put* transformations, under certain conditions, from the *get* transformation.

Simple example: expressions

- Concrete syntax: E_c, T_c, F_c

$$add_c : E_c \rightarrow E_c \text{ `}+\text{` } T_c \qquad sub_c : E_c \rightarrow E_c \text{ `}-\text{` } T_c$$

$$et_c : E_c \rightarrow T_c$$

$$mul_c : T_c \rightarrow T_c \text{ `}*\text{` } F_c \qquad tf_c : T_c \rightarrow F_c$$

$$nest_c : F_c \rightarrow \text{`(` } E_c \text{ `)`}$$

$$const_c : F_c \rightarrow Int_t$$

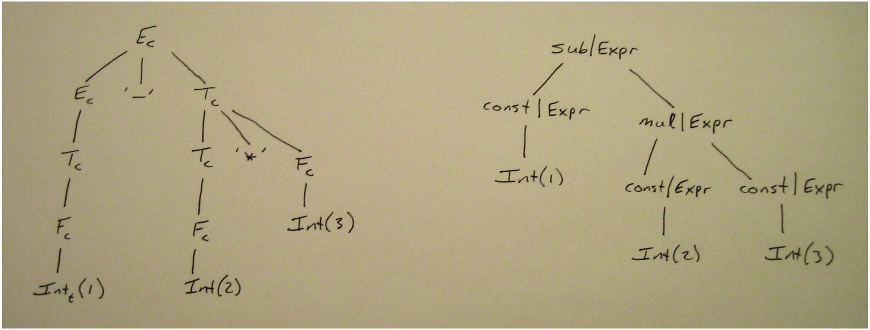
- Abstract syntax: $Expr$

$$add : Expr \rightarrow Expr\ Expr \qquad sub : Expr \rightarrow Expr\ Expr$$

$$mul : Expr \rightarrow Expr\ Expr$$

$$const : Expr \rightarrow Int_t$$

Consider the trees for "`1 - 2 * 3`"



get and *put* map back and forth

(Some) attributes for bx

- $getExpr$ :: $Expr$ occurs on E_c, T_c, F_c
- $putE_c$ :: E_c occurs on $Expr$

Specification for *get*

$$add_c : a :: E_c \rightarrow e :: E_c \text{ `+` } t :: T_c$$

$$a.getExpr = add\ (\ e.getExpr,\ t.getExpr\ )$$

$$sub_c : s :: E_c \rightarrow e :: E_c \text{ `}-\text{` } t :: T_c$$

$$s.getExpr = sub\ (\ e.getExpr,\ t.getExpr\ )$$

$$et_c : e :: E_c \rightarrow t :: T_c$$

$$e.getExpr = t.getExpr$$

$$mul_c : m :: T_c \rightarrow t :: T_c \text{ `}*\text{` } f :: F_c$$

$$m.getExpr = mul\ (\ t.getExpr,\ f.getExpr\ )$$

$$neg_c : n :: F_c \rightarrow \text{`}-\text{` } e :: E_c$$

$$n.getExpr = sub\ (\ const('0'),\ e.getExpr\ )$$

Naive specification for *put*

Consider a naive, handwritten implementation.

$add : a :: Expr \rightarrow l :: Expr \;\; r :: Expr$

$a.putE_c = add_c \; (\; l.getE_c, \text{`}+\text{`}, \;\; tf_c(nest_c(r.getE_c \;)))$

$sub : s :: Expr \rightarrow l :: Expr \;\; r :: Expr$

$s.putE_c = sub_c \; (\; l.getE_c, \text{`}-\text{`}, \;\; tf_c(nest_c(r.getE_c \;)))$

$mul : m :: Expr \rightarrow l :: Expr \;\; r :: Expr$

$m.putE_c = et_c(mul_c \; (\; tf_c(nest_c(l.getE_c)), \text{`}*\text{`}, \;\; nest_c(r.getE_c \;)))$

put(get("1 + 2 * 3")) = "1 + ((2) * (3))"

put(get("1 + - 2")) = "1 + (0 - (2))"

Obviously, this fails.

- We add too many parenthesis.
  Can *mul* map back to a $T_c$instead ?
- We don't map back to specialized cases.
  neg_c is not used.
  Or if we examine the target we may use neg_c when it wasn't in the source.

Having links back to the source would help.

Mapping back to multiple source NTs would help.

1. Linking from the target to the source.

   Reference attributes, forwarding, new productions.

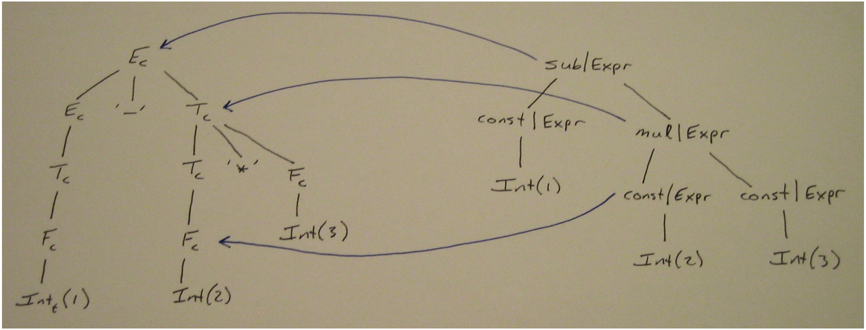
2. Multiple mappings back to the source.

   Several *put* attributes in the target AG.

Writing all of this is cumbersome, can we generate it?

Yes, sometimes.

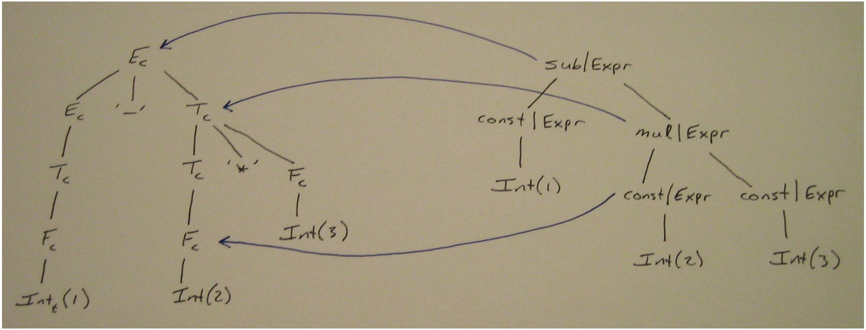
Links from the target to the source.



Reference Attributes

- attributes that are "pointers" to other tree nodes.
- we need to associate these pointers with nodes (productions) in the abstract syntax

Links are reference attributes



Forwarding

- Using inherited attributes to establish this links is unsatisfactory.
- Forwarding is used in (new) productions that
  - define some semantics (attributes), and
  - get remaining attributes from the "forwards to" tree
- *e.g.*

  $map: \ m :: Expr \longrightarrow \text{'}map\text{'} \ f :: Expr \ \ d :: Expr$

  $\{ \ m.errors = ...$

  forwards to ... translation to C ...

  $\}$
- similar to inheritance, but dynamic in nature

Forwarding in bx transformations

- Create a new production that takes the reference as a child and forwards to the original AST production / construct in the get specification.
- $sub_c : s :: E_c \longrightarrow e :: E_c$ '$-$' $t :: T_c$

  $s.getExpr = sub\_bx\_E_c \ (\ e.getExpr, \ t.getExpr, \ s \ )$

  {- from $\quad s.getExpr = sub(\ e.getExpr, \ t.getExpr \ )$ -}

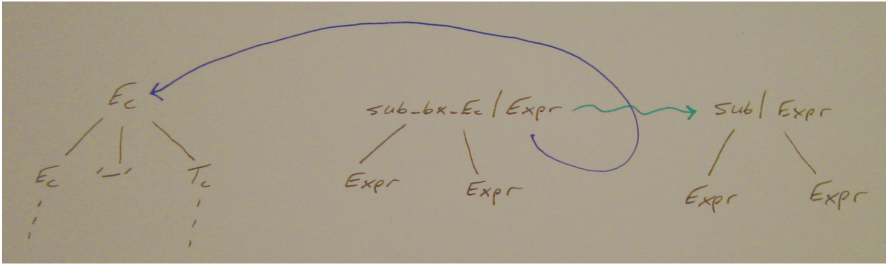
  $sub\_bx\_E_c : e :: Expr \longrightarrow l :: Expr \quad r :: Expr \quad s :: Ref \ E_c$

  { $e.putE_c =$ ... consider s ...

  forwards to $sub(l, \ r)$

  }

Links are reference attributes



Computing *put* on the forwarding production

- If only one concrete production maps to *sub* we know what the *put* transformation maps a *sub* to.
- But we may still use the source to access original terminal symbols.

$$sub\_bx\_E_c : e :: Expr \rightarrow l :: Expr \quad r :: Expr \quad s :: Ref \ E_c$$

$$\{ \quad e.putE_c = case \ s \ of$$

$$sub_c \ (\_, \ dash, \ \_) \rightarrow sub_c(l.putE_c, dash, r.putT_c)$$

$$\_ \ \rightarrow error(\ cannot \ get \ here \ )$$

$$\text{forwards to } sub(l, \ r) \quad \}$$

- We can pattern match on the source to
  - get access to terminals/layout in the source
  - determine the productions to use

Computing *put* on the forwarding production (2)

- Consider adding the contrived production

$subFrom_c : s :: E_c \rightarrow \text{`}subtract\text{`}\;\; e :: E_c \;\; \text{`}from\text{`}\;\; t :: T_c$

$$s.getExpr = sub\ (\ e.getExpr,\ \ t.getExpr\ )$$

- We extend the case expression on *sub* to

$sub\_bx\_E_c : e :: Expr \rightarrow l :: Expr \quad r :: Expr \quad s :: Ref\ \ E_c$

$\{\ \ e.putE_c = case\ \ \ s\ of$

$\quad sub_c\ (\_,\ \ dash,\ \ \_) \rightarrow sub_c(l.putE_c, dash, r.putT_c)$

$\quad subFrom_c\ (st, \_,\ \ fr,\ \ \_)$

$\qquad\qquad \rightarrow subFrom_c(st, l.putE_c, fr, r.putT_c)$

$\quad \_\ \ \rightarrow error(\ cannot\ get\ here\ )$

forwards to $sub(l,\ \ r)\quad \}$

Computing *put* on the forwarding production (3)

- In general, we add a production for each source NT type (E_c, F_c) that has a production (sub_c, $subFrom_c$ for $E_c$and neg_c for F_c) that translates to the target construct (sub).
- Other options
  - a new production for each source production $sub\_bx\_sub_c$, $sub\_bx\_subFrom_c$, $sub\_bx\_neg_c$
  - a single new production for each target production sub_bx

Generating the *put* definition

- *put* is defined by a case-expression that pattern matches on the source link.
- For

  $sub_c : s :: E_c \to e :: E_c \text{ `}-\text{` } t :: T_c$

  $\quad s.getExpr = sub\ (\ e.getExpr,\ t.getExpr\ )$

  this requires that *get* defined as a simple application of target language productions

  and that all components in the source, except constant terminals (keywords, punctuation, etc), provide pieces for the *get* translation.
- But can be extended to if-then-else constructs (Yellin).
- Limiting for general bx transformations but OK here.

Multiple mappings back to the source.

- $getExpr$ :: $Expr$ occurs on E_c, T_c, F_c
- $putE_c$ :: E_c occurs on $Expr$
  $putT_c$ :: T_c occurs on $Expr$
  $putF_c$ :: F_c occurs on $Expr$
- Above is generated from $nt :: \{E_c, T_c, F_c\} \rightarrow \{ Expr\}$
  $e.g.\ nt(E_c) = Expr$
  nt is a function
  nt', the inverse, is not
- In general, if $nt(A) = B$ generate
  $putA$ :: A occurs on B

Generating attribute definitions for *put*

For the new abstract productions (*e.g.* $sub\_bx\_putE_c$),

there are 2 cases for *putA*, for some NT A.

1. *putA* is defined by pattern matching on the source.
2. From other *put* attributes on the production.

The “can be” relation

- In the expression example
  - T_c can be E_c via et_c
  - F_c can be T_c via tf_c
  - E_c can be F_c via $nest_c$
- A source NT X “can be” a source NT Y if

  there is a production with X on the l.h.s and Y on the r.h.s.

  and a “copy rule” defines *get* on X to be *get* on Y.
- *e.g.*

  $et_c :\ e :: E_c \longrightarrow t :: T_c$

  $$e.getExpr = t.getExpr$$

$$sub\_bx\_E_c: \; e :: Expr \longrightarrow l :: Expr \quad r :: Expr \quad s :: Ref \;\; E_c$$

$$\{ \;\; e.putE_c = case \;\; s \; of$$

$$sub_c \; (\_, \; dash, \; \_) \longrightarrow sub_c(l.putE_c, dash, r.putT_c)$$

$$subFrom_c \; (st, \_, \; fr, \; \_)$$

$$\longrightarrow subFrom_c(st, l.putE_c, fr, r.putT_c)$$

$$\_ \;\; \longrightarrow error(\; cannot \;\; get \;\; here \;)$$

$$e.putT_c = tf_c(e.putF_c)$$

$$e.putF_c = nest_c(e.putE_c)$$

forwards to $sub(l, \; r)$

$$\}$$

Comments

- At least one of the *put* attributes needs to be defined explicitly - not based in the "can be" relation.
  AG circularity check should detect when this is not the case.

When the source link is absent

- We still need to define the put attributes on the original abstract syntax productions.

  e.g. . define $putE_c$, $putT_c$, and $putF_c$ on *sub*, *add*, *mul*.
- $mul : e :: Expr \rightarrow l :: Expr \quad r :: Expr$

  $\{ \quad e.putE_c = et_c(e.putT_c)$

  $\quad e.putT_c = mul_c(l.putT_c, '*', r.putF_c)$

  $\quad e.putF_c = nest_c(e.putE_c) \quad \}$
- $add(\text{“1”}, mul(\text{“2”}, \text{“3”})).putE_c \Longrightarrow$ “1 + 2 * 3”
- $mul(\text{“1”}, add(\text{“2”}, \text{“3”})).putE_c \Longrightarrow$ “1 * (2 + 3)”
- w/o source access $put(get(\text{“}1 + (2 * 3)\text{''})) \Longrightarrow \text{“}1 + 2 * 3\text{''}$
- All comes from the grammar - no specification of operator precedence or associativity.

Gory details

$$add : e :: Expr \rightarrow l :: Expr \quad r :: Expr$$

$$\{ \quad e.putE_c = add_c(l.putE_c, r.putT_c)$$

$$e.putT_c = tf_c(e.putF_c)$$

$$e.putF_c = nest_c(e.putE_c) \quad \}$$

$$mul : e :: Expr \rightarrow l :: Expr \quad r :: Expr$$

$$\{ \quad e.putE_c = et_c(e.putT_c)$$

$$e.putT_c = mul_c(l.putT_c, '*', r.putF_c)$$

$$e.putF_c = nest_c(e.putE_c) \quad \}$$

Grammar-module-aware pattern matching

- A pattern in a case-expression in the target language will not match these generated productions.
- We modify pattern matching such that on a tree constructed by productions not defined in the grammar of the case expression we match what that tree forwards to.

Future, ongoing work

- expand class of get transformation specs that we can generate put from
- support *get* transformations from which we can "almost" generate the complete *put* transformation.
- maintain layout: white space, comments
- fully incorporate into Silver, as an extension
- evaluation on the data-intensive applications
- Thanks for your attention.