Bidirectional Grammar Transformations

Software Languages Team, Universität Koblenz-Landau
Vadim Zaytsev, SWAT, CWI

CC BY-NC-SA 2012
In a nutshell

- Grammars in the broad sense
  - protocols, metamodels, schemata, ontologies, structures
- Grammars evolve
  - language evolution, error fixes, dialects, etc
- Systematic way
  - programmable transformation steps
  - operator suite
  - bidirectionality
Motivation
Why transformation?

- Edit as text
  - no (technical) problems during editing
  - some problems afterwards
  - no automation
  - no replication

- Program transformations
  - somewhat more complex technically
  - more maintainable, reproducible, transformable, …
Why bidirectional?

• Sweet spot between
  • functional approach
    • \( y = f(x) \)
  • declarative approach
    • \( f(x,y) :- \ldots \)

• Fixed application order, direction is flexible
Bidirectionality profits

(a) Different ways to represent the same relationship between grammars $G$ and $G'$ with unidirectional grammar transformation steps. (b) Different possible transformation graphs for converging $G_1$, $G_2$, and $G_3$.

Using unidirectional transformations obfuscates the actual relationship somewhat since one relationship can be represented by a set of different transformation scenarios. See Figure qn Hai in Deploying a bidirectional transformation instead will solidify the relationship in a way that can be reversed or partially reversed if necessary during runtime.

Mapping between syntactic notations. When a language engineer possesses a number of grammars (grammar base) specified by means of a particular syntactic notation $\text{rw}$ they may need to be altered if there is an intention to use a particular grammarware framework (say, GDK $\text{v}$, TXL $\text{rx}$, Rascal $\text{ry}$, SLPS $\text{sp}$) that works with a different yet perhaps even equivalent notation. Bidirectionality plays an important role here because if the grammarware framework changes the grammar, such changes will need to be propagated back to the original notation. In $\text{sq}$ we have already raised a question of expressing language evolution as bidirectional grammar transformation sequences in the context of syntactic notation change.

Directed transformation graph transformation. Complex transformation scenarios may often be understood as directed acyclic graphs with nodes corresponding to grammars and edges corresponding to grammar transformations. See Figure qn hbi in Directed transformation graphs. One strong instance of this notion is grammar convergence with the graph being a tree with grammars as nodes and grammar transformations as edges. Rearranging such transformation graphs (e.g., adjusting the convergence tree) or changing their topology (e.g., deriving a coevolution scenario from a convergence tree) is difficult unless certain important graph refactorings are effectively performed. This can be achieved through bidirectional transformation - C Wiley.
Unidirectional Transformations
Transformation components
Transformation components

- known semantics, well-defined algorithm
- rename, fold, factor, inject, remove, …
Transformation components

Arguments

• what exactly to rename/factor/inject/…?
Transformation components

Input grammar
- determines applicability
Variations on components

Variations possible

A fragment of concrete syntax.
What if we want to derive the abstract syntax?
Example

A fragment of concrete syntax.
What if we want to derive the abstract syntax?

expr : ...;
atom : ID | INT | '(' expr ')';

Need to project away ‘(’ & ‘)’

Need to merge “expr” & “atom”

Alternative needs to go entirely
A transformation sequence

expr : ...;
atom : ID | INT | '( expr )';

abstractize

expr : ...;
atom : ID | INT | expr;

vertical

expr : ...;
atom : ID;
atom : INT;
atom : expr;

unite

expr : ...;
expr : ID;
expr : INT;

abridge

expr : ...;
expr : INT;
expr : expr;
XBGF Operator Suite

- Semantics-preserving (refactoring)
  - rename, import, introduce, eliminate
  - fold, unfold, extract, inline
  - factor, distribute, horizontal, vertical
  - yaccify, deyaccify, massage
  - designate, unlabel
  - abridge, detour
  - ...

$L(G_1) = L(G_2)$
XBGF Operator Suite

- Semantics-in/decreasing
  - appear, disappear
  - narrow, widen
  - add, remove
  - upgrade, downgrade
  - unite
  - ...

\[
L(G_1) \subseteq L(G_2) \quad \lor \quad L(G_2) \subseteq L(G_1)
\]
XBGF Operator Suite

• Semantics-revising
  - undefined, define
  - inject, project, permute
  - abstractize, concretize
  - replace, redefine

\[ L(G_1) \not\subseteq L(G_2) \]
\[ \land \]
\[ L(G_2) \not\subseteq L(G_1) \]
Bidirectional Transformations
EBGF Operator Suite

- Pairs already formed
  - chain — unchain
  - add — remove
  - fold — unfold
  - …

- Implicit pairs
  - factor — factor
  - massage — massage
  - replace — replace
  - …
EBGF Operator Suite

- Incomplete pairs
  - rassoc — ???
  - unite — ???
  - equate — ???
  - ...

- Asymmetrical pairs
  - designate(prod) — unlabel(label)
  - deyaccify(nt) — yaccify(prods)
  - eliminate(nt) — introduce(prods)
  - ...

EBGF Operator Suite

- Remaining problems
  - Lack of symmetry in execution
    - unfold(A:ε) — fold???
    - unfold(A:b in bAb) — fold???
  - ...
- Hence, bidirectionality and not bijection
To summarise

- Grammar transformation steps describe software language evolution
- Programmable grammar transformation operators
- ΞBGF: unidirectional
- ΞBGF: bidirectional
- SLPS: implementations in Prolog and Rascal
- http://grammarware.github.com
Questions?

vadim@grammarware.net