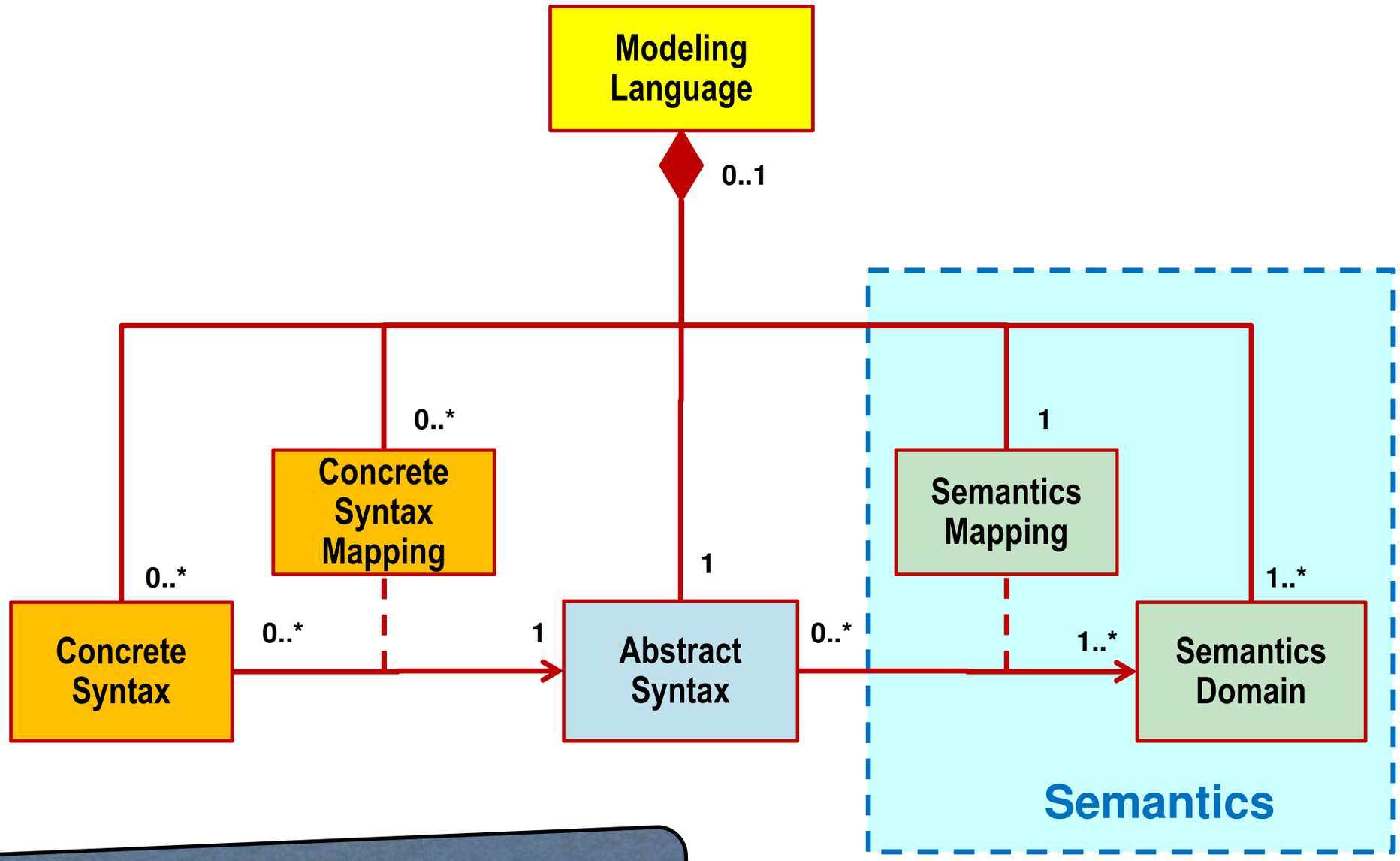
Language modeling principles

Ralf Lämmel
Software Language Engineer, University of Koblenz-Landau

Part of the MODELS'14 tutorial on
"Language Modeling Principles"
http://softlang.uni-koblenz.de/models14/

Elements of a (Modeling) Language

Source: Bran Selic: “The “Theory” and Practice of Modeling Language Design, Tutorial at MODELS’13.



This is a principle of Modeling Language Modeling.

Let's model any sort of language!

- "Modeling" as in "(executable) definition"
  - Define abstract and concrete syntax.
    - Leverage those for parsing, unparsing, and editing.
  - Define semantics of a language.
    - Leverage those for interpretation and transformation.
  - Define properties for language.
    - Well-* properties, static / dynamic analysis properties.
      - Leverage those for the benefit of language user.

Technological spaces in scope

- Grammarware
  - Rascalware
- Programware
  - Funware
    - Haskellware
  - Logicware
    - Prologware
  - Objectware
    - Javaware
- Dataware
  - Sqlware
- Modelware
  - Emfware
- Ontoware

Venues in scope

- OOPSLA: Object-Oriented Programming, Systems, Languages, and Applications
- MODELS: Model Driven Engineering Languages and Systems
- GPCE: Generative Programming: Concepts & Experiences
- PLDI: Programming Language Design and Implementation
- PEPM: Partial Evaluation and Program Manipulation
- SCAM: Source Code Analysis and Manipulation
- POPL: Principles of Programming Languages
- SLE: Software Language Engineering
- CC: Compiler Construction

Topics in this tutorial

- Representation formats
- Basic modeling tasks
- Models of computation
- Pretty printing
- Parsing text to trees
- Megamodeling (UEBERmodeling)
- Software transformations
- Reference resolution
- (Structure editing)
- The software ontology SoLaSoTe

Prolog as the metalanguage

Languages in this tutorial

- *expr, figure, family*: simple ***sample*** languages
- *dict, graph*: languages of ***dictionaries*** and ***graphs***
- *bsl*, *esl*: basic and extended ***signature*** language
- *bgl*, *egl*: basic and extended ***grammar*** language
- *ddl*: ***data definition*** language (SQL subset)
- *mml*: ***metamodeling*** language
- *ppl*: ***pretty printer*** language
- *html*: obvious
- *ueber*: a megamodeling language

Prolog as the metalanguage

Format of this tutorial

- Objective:
  - Collect language modeling principles.
  - Discuss language modeling education.
- Format:
  - Prolog is used for illustration.
  - Please interrupt at any time!
  - Let's discuss a lot.

Ralf Lämmel
@reallynotabba

Material of today's #models14 tutorial on "language modeling principles" are available here: softlang.uni-koblenz.de/models14/

29/09/14 07:36

http://softlang.uni-koblenz.de/models14/

Detour I/II:

A Software Language Engineering course

http://softlang.wikidot.com/course:sle

Detour II/II:

A nonsystematic literature survey for SLE

http://softlang.uni-koblenz.de/yabib.pdf

Topics in this tutorial

- **Representation formats**
- Basic modeling tasks
- Models of computation
- Pretty printing
- Parsing text to trees
- Megamodeling (UEBERmodeling)
- Software transformations
- Reference resolution
- (Structure editing)
- The software ontology SoLaSoTe

Representation formats

Ralf Lämmel
Software Language Engineer, University of Koblenz-Landau

Part of the MODELS'14 tutorial on
"Language Modeling Principles"
http://softlang.uni-koblenz.de/models14/

Representation formats

- (Textual syntax)
- (Visual syntax)
- Algebraic terms
- Key-value maps (dictionaries)
- ... with ids and references (graphs)

Prolog as the metalanguage

Textual syntax of the *expr* language

- Intuitively
  - **0 + 1**
- Actual textual syntax
  - **zero + succ(zero)**

Online: languages/expr

Prefix term

```
add(
  const(zero),
  const(succ(zero))
).
```

Online: languages/expr

(Nested) dictionary

```
{
 class : add,
 left : {
        class : const,
  value : {
        class : zero
        }
  },
 right : {
        class : const,
        value : {
              class : succ,
              pred : {
                    class : zero
              }
        }
  }
}.
```

Online: languages/expr

Graph (used here for sharing)

```
{
 class : add,
 left : {
  class : const,
  value : 0 & {
          class : zero
         }
},
right : {
         class : const,
         value : {
          class : succ,
          pred : # 0
         }
        }
}.
```

Online: languages/expr

Details on terms

Prefix terms

```
% A term consists of a symbol and a list of subterms.
prefixTerm(Term) :-
  Term =.. [Symbol|Terms],
  atom(Symbol),
  map(prefixTerm, Terms).
```

Online: languages/bsl/prefix-term.pro

(Basic) signature for *expr*

```
% A signature for simple expressions
signature(

 % Sorts of this signature
  [nat, expr],

  % Symbol types
  [ symbol(zero, [], nat), % zero ("0")
    symbol(succ, [nat], nat), % successor function
    symbol(const, [nat], expr), % a number is an expression
    symbol(add, [expr, expr], expr) % binary addition
  ]
).
```

Online: languages/expr/as.term

Conformance with a signature

```
% Many-sorted terms for a given signature
manySortedTerm(Sig, Term, Sort) :-
  signature(Sig),
  prefixTerm(Term),
  Sig = signature(Sorts, Profiles),
  member(Sort, Sorts),
  manySortedTerm_(Profiles, Term, Sort).

% Recursive test for term symbols to comply with a type
manySortedTerm_(Profiles, Term, Result) :-
  Term =.. [Symbol|Terms],
  member(symbol(Symbol, Arguments, Result), Profiles),
  map(manySortedTerm_(Profiles), Terms, Arguments).
```

Online: languages/bsl/conformance.pro

A higher-order bit

```
% All list elements must meet a certain
predicate.
map(_, []).
map(G, [H|T]) :-
  apply(G, [H]), map(G, T).

% Map a function-like predicate over a list
map(_, [], []).
map(P, [H1|T1], [H2|T2]) :-
  apply(P, [H1, H2]), map(P, T1, T2).

% Another cardinality for map
map(_, [], [], []).
map(P, [H1|T1], [H2|T2], [H3|T3]) :-
  apply(P, [H1, H2, H3]), map(P, T1, T2, T3).
```

Online: prelude/higher-order.pro

In need of applied terms
(Examples based on “figure” language)

Concrete syntax:

```
line from: (0, 0), to: (0, 4);
line from: (0, 4), to: (3, 5);
line from: (3, 5), to: (0, 0);
```

Abstract syntax (prefix term format):

```
[
  line((0, 0), (0, 4)),
  line((0, 4), (3, 5)),
  line((3, 5), (0, 0))
].
```

Online: languages/figure

(Untyped) applied terms

```
% Applied terms also covering lists and pairs
appliedTerm(Term) :-
  Term =.. [Symbol|Terms],
  atom(Symbol),
  map(appliedTerm, Terms).

% Integers as applied terms
appliedTerm(Term) :-
  integer(Term).
```

Online: languages/esl/applied-term.pro

(Applied) signature for *figure*

```
[
  line((0, 0), (0, 4)),
  line((0, 4), (3, 5)),
  line((3, 5), (0, 0))
].
```

Type aliases

Lists

```
[
  type(figure, star(sort(element))),
  symbol(line, [sort(point), sort(point)], element),
  symbol(circle, [sort(point), integer], element),
  type(point, tuple([integer, integer]))
].
```

Constructors

Tuples

Online: languages/figure/as.term

Signature of signatures

Excerpt

```
...

% Different kind of term types
symbol(term, [], typeexpr), % untyped terms
symbol(atom, [], typeexpr), % primitive type "atom"
symbol(integer, [], typeexpr), % ... "integer"
symbol(float, [], typeexpr), % ... "float"
symbol(number, [], typeexpr), % ... "number"
symbol(boolean, [], typeexpr), % ... "boolean"
symbol(sort, [atom], typeexpr), % sort reference
symbol(star, [sort(typeexpr)], typeexpr), % list types
symbol(plus, [sort(typeexpr)], typeexpr), % list types
symbol(option, [sort(typeexpr)], typeexpr), % option types
symbol(tuple, [star(sort(typeexpr))], typeexpr) % tuple types

...
```

Online: languages/esl/as.term

Conformance with a signature

Excerpt

```
% Apply symbol
wellTypedTerm_(Decls, sort(Result), Term) :-
  Term =.. [Symbol|Terms],
  member(symbol(Symbol, Arguments, Result), Decls),
  map(wellTypedTerm_(Decls), Arguments, Terms).

% Apply alias
wellTypedTerm_(Decls, sort(Sort), Term) :-
  member(alias(Sort, Type), Decls),
  wellTypedTerm_(Decls, Type, Term).
```

Online: languages/esl/conformance.pro

Additional constraints on top of self-description

Excerpt

```
...

% No double declarations of symbols
\+ (
  member(symbol(Symbol, Arguments1, Result1), Decls),
  member(symbol(Symbol, Arguments2, Result2), Decls),
  \+ (
    Arguments1 == Arguments2,
    Result1 == Result2
  )
),

...
```

Online: languages/esl/as.pro

Reflections

- Representation formats team up with “types”.
  - Terms team up with algebraic signatures.
  - EMF models team with EMF metamodels.
  - ...
- Self-representation attests expressiveness.
  - Signature of signatures.
  - Metametamodel in MDE.
  - ...
- More to come on dictionaries and graphs.

Topics in this tutorial

- Representation formats
- **Basic modeling tasks**
- Models of computation
- Pretty printing
- Parsing text to trees
- Megamodeling (UEBERmodeling)
- Software transformations
- Reference resolution
- (Structure editing)
- The software ontology SoLaSoTe

Basic modeling tasks

Ralf Lämmel
Software Language Engineer, University of Koblenz-Landau

Part of the MODELS'14 tutorial on
"Language Modeling Principles"
http://softlang.uni-koblenz.de/models14/

Basic modeling tasks

- Define examples.
- Define concrete syntax.
  - ... as suitable for parsing or editing etc.
- Define abstract syntax (w/o constraints).
- Define mapping between syntaxes.
- Define wellness.
- Define semantics.

All definitions are to be executable.

The *yafpl* language

```
factorial :: Int -> Int
factorial x =
  if ((==) x 0)
    then 1
    else ((*) x (factorial ((-) x 1)))
```

- YAFPL - Yet another functional programming language
- A syntactical subset of Haskell (see above)

Online: languages/yafpl

Concrete syntax of *yafpl*

```
program : { function } * ;
function : funsig fundef ;
funsig : name '::' type ;
fundef : name { name }* '=' expr ;
type : simpletype { '->' simpletype }* ;

[inttype] simpletype :  'Int' ;
[booltype] simpletype :  'Bool' ;

[op] expr : '(' op ')' subexpr subexpr ;
[subexpr] expr : subexpr ;
[apply] expr : name { subexpr }+ ;
[intconst] subexpr : int ;
[brackets] subexpr : '(' expr ')' ;
[if] subexpr : 'if' expr 'then' expr 'else' expr ;
[name] subexpr : name ;

[add] op : '+' ;
[sub] op : '-' ;
[mult] op : '*' ;
[eq] op : '==' ;
```

Lexical syntax of *yafpl*

```
name : lower { alpha }* ;
int : { digit }+ ;
layout : { space }+ ;
```

Abstract syntax of *yafpl*

```
[
 (
  (factorial,[inttype],inttype),
  factorial,
  [x],
  if(
   op(eq,name(x),intconst(0)),
   intconst(1),
   op(mult,
    name(x),
    apply(factorial,[op(sub,name(x),intconst(1))])
   )
  )
 )
]
```

Signature of *yafpl* abstract syntax

```
signature(
  [
    alias(program, list(sort(function))),
    alias(function, tuple([
      sort(funsig),
      sort(fundef)])),
    alias(funsig, tuple([
      sort(name),
      list(sort(simpletype)),
      sort(simpletype)])),
    alias(fundef, tuple([
      sort(name),
      list(sort(name)),
      sort(expr)])),
    symbol(inttype, [], simpletype),
    symbol(booltype, [], simpletype),
    symbol(intconst, [integer], expr),
    symbol(boolconst, [boolean], expr),
    symbol(name, [sort(name)], expr),
    symbol(if, [sort(expr), sort(expr), sort(expr)], expr),
    symbol(op, [sort(op), sort(expr), sort(expr)], expr),
    symbol(apply, [sort(name), list(sort(expr))], expr),
    symbol(add, [], op),
    symbol(sub, [], op),
    symbol(mult, [], op),
    symbol(eq, [], op),
    alias(name, atom)
  ]
).
```

Online: languages/yafpl

Mapping *yafpl* concrete to abstract syntax

```
% Mapping for function types
yafplMapping(type, (T1, Ts1), ([T1|Ts2], T2)) :-
  append(Ts2, [T2], Ts1).

% Eliminate layering in expressions
yafplMapping(expr, subexpr(E), E).

% Eliminate brackets in expressions
yafplMapping(subexpr, brackets(E), E).

% Lexical mapping for int
yafplMapping(int, Digits, Int) :-
  number_codes(Int, Digits).

% Lexical mapping for name
yafplMapping(name, (Char, String), Atom) :-
  name(Atom, [Char|String]).
```

Wellness for *yafpl* 1/5

```
% Wellness of collection of function declarations
okProg(P) :-
  map(toFunName, P, Ns),
  set(Ns),
  map(okFun(P), P).

toFunName(((N, _, _), _), N).
```

Wellness for *yafpl* 2/5

```
% Wellness of function declarations
okFun(P, ((N, Ts, T), (N, Ns, E))) :-
    set(Ns),
    zip(Ns, Ts, X),
    okExpr(P, X, E, T).
```

Wellness for *yafpl* 3/5

```
% An int constant is of the int type
okExpr(_, _, intconst(_), inttype).

% The context provides the type of a variable
okExpr(_, X, name(N), T) :-
  member((N, T), X).

% Condition is of boolean type; others are of the same type
okExpr(P, X, if(E1, E2, E3), T) :-
  okExpr(P, X, E1, booltype),
  okExpr(P, X, E2, T),
  okExpr(P, X, E3, T).
```

Wellness for *yafpl* 4/5

```
% Check operator application
okExpr(P, X, op(O, E1, E2), T0) :-
  okExpr(P, X, E1, T1),
  okExpr(P, X, E2, T2),
  okOp(O, T1, T2, T0).

% Check function application
okExpr(P, X, apply(F, Es), T) :-
  map(okExpr(P, X), Es, Ts),
  member(((F, Ts, T), _), P).
```

Wellness for *yafpl* 5/5

```
% Operand types of operators
okOp(add, inttype, inttype, inttype).
okOp(sub, inttype, inttype, inttype).
okOp(mult, inttype, inttype, inttype).
okOp(eq, inttype, inttype, booltype).
```

Big-step semantics of *yafpl* 1/4

```
value(intval(_)).
value(boolval(_)).
```

Big-step semantics of *yafpl* 2/4

```
% A constant evaluates to itself
evaluate(_, _, intconst(I), intval(I)).

% A variable evaluates to its binding
evaluate(_, X, name(N), V) :-
  member((N, V), X).

% True condition
evaluate(P, X, if(E1, E2, _), V) :-
  evaluate(P, X, E1, boolval(true)),
  evaluate(P, X, E2, V).

% False condition
evaluate(P, X, if(E1, _, E2), V) :-
  evaluate(P, X, E1, boolval(false)),
  evaluate(P, X, E2, V).
```

Big-step semantics of *yafpl* 3/4

```
% Evaluate operator application
evaluate(P, X, op(O, E1, E2), V0) :-
  evaluate(P, X, E1, V1),
  evaluate(P, X, E2, V2),
  opVal(O, V1, V2, V0).

% Evaluate function application
evaluate(P, X1, apply(N, Es), V) :-
  map(evaluate(P, X1), Es, Vs),
  member((_, (N, Ns, E)), P),
  zip(Ns, Vs, X2),
  evaluate(P, X2, E, V).
```

Big-step semantics of *yafpl* 4/4

```
opVal(add, intval(I1), intval(I2), intval(I0)) :-
  I0 is I1 + I2.

opVal(sub, intval(I1), intval(I2), intval(I0)) :-
  I0 is I1 - I2.

opVal(mult, intval(I1), intval(I2), intval(I0)) :-
  I0 is I1 * I2.

opVal(eq, intval(I1), intval(I2), boolval(B0)) :-
  toBoolean((I1==I2), B0).
```

Reflections

- We picked a ***textual*** concrete syntax.
- We enabled ***parsing*** for concrete syntax.
- We picked a ***term*** domain for abstract syntax.
- We sufficed with ***type checking*** (no inference).
- We picked a ***big-step*** semantics.
- No translation covered.

A more advanced example:

FSML
= FSM Language
= Finite State Machine Language

Specification: [.pdf]
Source code: [GitHub]

Topics in this tutorial

- Representation formats
- Basic modeling tasks
- **Models of computation**
- Pretty printing
- Parsing text to trees
- Megamodeling (UEBERmodeling)
- Software transformations
- Reference resolution
- (Structure editing)
- The software ontology SoLaSoTe

Models of computation

Ralf Lämmel
Software Language Engineer, University of Koblenz-Landau

Part of the MODELS'14 tutorial on
"Language Modeling Principles"
http://softlang.uni-koblenz.de/models14/

Motivation

- Big-step semantics is "opaque" on computation.
  - How to debug a program?
  - How to make partial progress (c.f. parallelism)?
  - How to weave with another semantics?
- Let us examine small-step semantics, thus!
  - Additional fundamental notions:
    - Normal form (value)
    - Substitution

Normal form of expression evaluation

```
...
symbol(intconst, [integer], expr),
symbol(boolconst, [boolean], expr),
symbol(name, [sort(name)], expr),
symbol(if, [sort(expr), sort(expr), sort(expr)], expr),
symbol(op, [sort(op), sort(expr), sort(expr)], expr),
symbol(apply, [sort(name), list(sort(expr))], expr),
...
```

Online: languages/yafpl

Normal form of expression evaluation

```
normal(intconst(_)).
normal(boolconst(_)).
```

Online: languages/yafpl/small-step/normal.pro

(Stepwise) reduction

```
% Reflexive case
reduce(_, E, E) :-
  normal(E).

% Transitive case
reduce(P, E1, E3) :-
  step(P, E1, E2),
  reduce(P, E2, E3).
```

Online: languages/yafpl/small-step/reduction.pro

Step relation 1/3

```
% Simplify condition
step(P, if(E1a, E2, E3), if(E1b, E2, E3)) :-
  step(P, E1a, E1b).

% Commit to then branch
step(_, if(boolconst(true), E, _), E).

% Commit to then branch
step(_, if(boolconst(false), _, E), E).
```

Online: languages/yafpl/small-step/step.pro

Step relation 2/3

```
% Simplify left operand
step(P, op(O, E1a, E2), op(O, E1b, E2)) :-
  step(P, E1a, E1b).

% Simplify right operand
step(P, op(O, E1, E2a), op(O, E1, E2b)) :-
  normal(E1),
  step(P, E2a, E2b).

% Apply operator
step(_, op(O, E1, E2), E0) :-
  normal(E1),
  normal(E2),
  opConst(O, E1, E2, E0).
```

Online: languages/yafpl/small-step/step.pro

Step relation 3/3

```
% Simplify operand of function application
step(P, apply(F, Es1), apply(F, Es4)) :-
  append(Es2, [E1|Es3], Es1),
  map(normal, Es2),
  step(P, E1, E2),
  append(Es2, [E2|Es3], Es4).

% Apply function
step(P, apply(N, Es), E2) :-
  map(normal, Es),
  member((_, (N, Ns, E1)), P),
  zip(Ns, Es, L),
  star(substitute, L, E1, E2).
```

Online: languages/yafpl/small-step/step.pro

Substitution


```
substitute(_, intconst(I), intconst(I)).
substitute(_, boolconst(B), boolconst(B)).
substitute((N,E), name(N), E).
substitute((N1,_), name(N2), name(N2)) :- N1 \= N2.
substitute(S, if(E1a, E2a, E3a), if(E1b, E2b, E3b)) :-
  substitute(S, E1a, E1b),
  substitute(S, E2a, E2b),
  substitute(S, E3a, E3b).
substitute(S, op(O, Left1, Right1), op(O, Left2, Right2)) :-
  substitute(S, Left1, Left2),
  substitute(S, Right1, Right2).
substitute(S, apply(F, Es1), apply(F, Es2)) :-
  map(substitute(S), Es1, Es2).
```

Online: languages/yafpl/small-step/substitution.pro

A higher-order bit

```
% EBNF-like "*" for accumulating predicate
star(P, X, Y) :- plus(P, X, Y).
star(_, X, X).

% Extension of star/3 to add list construction
star(P, L, X, Y) :- plus(P, L, X, Y).
star(_, [], X, X).
```

Online: prelude/higher-order.pro

Topics in this tutorial

- Representation formats
- Basic modeling tasks
- Models of computation
- **Pretty printing**
- Parsing text to trees
- Megamodeling (UEBERmodeling)
- Software transformations
- Reference resolution
- (Structure editing)
- The software ontology SoLaSoTe

Pretty printing

Ralf Lämmel
Software Language Engineer, University of Koblenz-Landau

Part of the MODELS'14 tutorial on
"Language Modeling Principles"
http://softlang.uni-koblenz.de/models14/

Pretty printer development

- Pick the (abstract) syntax of input.
- Pick the (concrete) syntax of output.
- Set up test case(s).
- Define mapping from input to “boxes”.
- Evaluate “boxes” to obtain output.
- Validate pretty printer with test case(s).

The *yafpl* language

```
factorial :: Int -> Int
factorial x =
  if ((==) x 0)
    then 1
    else ((*) x (factorial ((-) x 1)))
```

- YAFPL - Yet another functional programming language
- A syntactical subset of Haskell (see above)

Online: languages/yafpl

Abstract syntax of *yafpl*

```
[
 (
  (factorial,[inttype],inttype),
  factorial,
  [x],
  if(
   op(eq,name(x),intconst(0)),
   intconst(1),
   op(mult,
    name(x),
    apply(factorial,[op(sub,name(x),intconst(1))])
   )
  )
 )
].
```

Online: languages/yafpl

Abstract syntax of *ppl* (“boxes”)

```
signature(
  [
    symbol(empty, [], box),
    symbol(text, [atom], box),
    symbol(hbox, [sort(box), sort(box)], box),
    symbol(hlist, [list(sort(box))], box),
    symbol(vbox, [sort(box), sort(box)], box),
    symbol(vlist, [list(sort(box))], box),
    symbol(indent, [sort(box)], box)
  ]
).
```

Online: languages/ppl

Pretty printer for *yafpl*

```
% Render conditionals
ppExpr(
  if(E1, E2, E3),
  vbox(
    hlist([text('if'), indent(B1)]),
    indent(indent(vbox(
      hbox(text('then'), indent(B2)),
      hbox(text('else'), indent(B3))
    )))
  )
) :-
    ppExpr(E1, B1),
    ppExpr(E2, B2),
    ppExpr(E3, B3).
```

Online: languages/yafpl

Evaluator of *ppl* 1/3

```
% Evaluate a text box
ppEval(text(A), [S]) :-
  atom_codes(A, S).

% Evaluate an empty box
ppEval(empty, []).

% Evaluate a vertical composition
ppEval(vbox(B1, B2), L3) :-
  ppEval(B1, L1),
  ppEval(B2, L2),
  append(L1, L2, L3).

% Vector form of vertical composition
ppEval(vlist(Bs), L) :-
  map(ppEval, Bs, Ls),
  foldr(append, [], Ls, L).
```

Online: languages/ppl

Evaluator of *ppl* 2/3

```
% Evaluate a horizontal composition
ppEval(hbox(B1, B2), L3) :-
  ppEval(B1, L1),
  ppEval(B2, L2),
  happend(L1, L2, L3).

% Vector form of horizontal composition
ppEval(hlist(Bs), L) :-
  map(ppEval, Bs, Ls),
  foldr(happend, [], Ls, L).

% Apply indentation
ppEval(indent(B), L) :-
  ppEval(hbox(text(' '), B), L).
```

Online: languages/ppl

Evaluator of *ppl* 3/3

```
% Horizontal composition of boxes (consisting of many lines)
happend(L1, L2, L3) :-
  map(length, L1, Lens),
  foldr(max, 0, Lens, Len),
  repeat(0' , Len, Spaces),
  happend(Spaces, L1, L2, L3).

% Helper for happend/3
happend(_, [], [], []).
happend(Spaces, [H1|T1], [H2|T2], [H3|T3]) :-
  append(H1, H2, H3),
  happend(Spaces, T1, T2, T3).
happend(_, [H1|T1], [], [H1|T1]).
happend(Spaces, [], [H2|T2], [H3|T3]) :-
  append(Spaces, H2, H3),
  happend(Spaces, [], T2, T3).
```

Online: languages/ppl

Some higher-order bits

```
% Left-associative list fold
foldl(_, U, [], U).
foldl(F, U, [H|T], Z) :-
  apply(F, [U, H, Y]),
  foldl(F, Y, T, Z).

% Right-associative list fold
foldr(_, X, [], X).
foldr(F, X, [H|T], Z) :-
  foldr(F, X, T, Y),
  apply(F, [H, Y, Z]).
```

Online: prelude/higher-order.pro

Reflections

- Our pretty printer contains boilerplate code.
  - Much of it could be generated.
- Priorities are not universally taken care of.
  - Declarative model of priorities needed (again).
- How does pretty printing compare to
  - templates and
  - “model-to-text”?

Topics in this tutorial

- Representation formats
- Basic modeling tasks
- Models of computation
- Pretty printing
- **Parsing text to trees**
- Megamodeling (UEBERmodeling)
- Software transformations
- Reference resolution
- (Structure editing)
- The software ontology SoLaSoTe

Parsing text to trees

Ralf Lämmel
Software Language Engineer, University of Koblenz-Landau

Part of the MODELS'14 tutorial on
"Language Modeling Principles"
http://softlang.uni-koblenz.de/models14/

The *expr* language

- Intuitively
  - **0 + 1**
- Actual textual syntax
  - **zero + succ(zero)**
- Abstract term-based syntax
  - **add(**
    const(zero),
    const(succ(zero))
)

Online: languages/expr

Signature of the *expr* language

```
signature(

  % Sorts of this signature
  [nat, expr],

  % Symbol types
  [ symbol(zero, [], nat), % zero ("0")
    symbol(succ, [nat], nat), % successor function
    symbol(const, [nat], expr), % a number is an expression
    symbol(add, [expr, expr], expr) % binary addition
  ]
).
```

Terminals of the *expr* language

```
['zero', 'succ', '(', ')', '+']
```

Scanner of the *expr* language

```
% A scanner for the expr language
exprScanner(Input, Ts) :-
  tokens(
    token(['zero', 'succ', '(', ')', '+']),
    Input,
    Ts).
```

Reusable token sequencer

- Arguments:
  - Token recognizer as on previous slide
  - Another token recognizer ("layout")
  - Input (a string to be completely consumed)
- Result:
  - A list of tokens such as

[zero, '+', succ, '(', zero, ')']

Online: prelude/scanning.pro

The *expr* grammar

```
[zero] nat : 'zero' ;
[succ] nat : 'succ' '(' nat ')' ;
[const] expr : nat ;
[add] expr : expr '+' expr ;
```

The *expr* grammar as a term

```
grammar(

  % Nonterminals of the grammar
  [nat, expr],

  % Production rules of the grammar
  [
    rule(zero, nat, [t(zero)]),
    rule(succ, nat, [t(succ), t('('), n(nat), t(')')]),
    rule(const, expr, [n(nat)]),
    rule(add, expr, [n(expr), t('+'), n(expr)])
  ]
).
```

The signature of (BGL) grammars

```
% The signature of context-free grammars
signature(
  [

    % Grammars as lists of nonterminals and rules
    symbol(grammar, [
        list(sort(nonterminal)), % list of nonterminals
        list(sort(rule)) % list of rules
      ],
      grammar),

    % Rules with LHS and RHS as well as a label
    symbol(rule, [
        sort(label), % label of rule
        sort(nonterminal), % LHS nonterminal
        list(sort(symbol)) % RHS sequence of symbols
      ],
      rule),

    % Classification of grammar symbols
    symbol(t, [sort(terminal)], symbol), % terminals are symbols
    symbol(n, [sort(nonterminal)], symbol), % nonterminals as well

    % Elementary kinds of symbols
    alias(nonterminal, atom),
    alias(terminal, atom),
    alias(label, atom)

  ]
).
```

Online: languages/bgl

BGL acceptor - top-down

```
% Accept input, non-deterministically and top-down
acceptTopDown(
    grammar(_, Rules), % rules to interpret
    Root, % root nonterminal
    Input % input string of terminals
  ) :-
       acceptTopDown_(Rules, [n(Root)], Input, []).

% Acceptance completed
acceptTopDown_(_, [], Input, Input).

% Consume terminal at top of stack from input
acceptTopDown_(
    Rules,
    [t(T)|Stack], % parser stack with terminal at the top
    [T|Input0], % input with ditto terminal as head
    Input1
  ) :-
       acceptTopDown_(Rules, Stack, Input0, Input1).

% Expand nonterminal at top of stack
acceptTopDown_(
    Rules,
    [n(N)|Stack0], % parser stack with nonterminal at the top
    Input0, Input1
  ) :-
       member(rule(_, N, Rhs), Rules),
       append(Rhs, Stack0, Stack1),
       acceptTopDown_(Rules, Stack1, Input0, Input1).
```

Online: languages/bgl

BGL acceptor - bottom-up

```
% Accept input, non-deterministically and bottom-up
acceptBottomUp(
    grammar(_, Rules), % rules to interpret
    Root, % root nonterminal
    Input % input string of terminals
  ) :-
      acceptBottomUp_(Rules, [], [n(Root)], Input).

% Acceptance completed
acceptBottomUp_(_, Stack, Stack, []).

% Shift terminal from input to stack
acceptBottomUp_(Rules, Stack0, Stack1, [T|Input0]) :-
  acceptBottomUp_(Rules, [t(T)|Stack0], Stack1, Input0).

% Reduce prefix of stack to according to rule
acceptBottomUp_(Rules, Stack0, Stack2, Input0) :-
  append(RhsReverse, Stack1, Stack0),
  reverse(RhsReverse, Rhs),
  member(rule(_, N, Rhs), Rules),
  acceptBottomUp_(Rules, [n(N)|Stack1], Stack2, Input0).
```

Online: languages/bgl

A parse tree

```
fork(
  rule(add, expr, [n(expr), t('+'), n(expr)]),
  [
    fork(
      rule(const, expr, [n(nat)]),
      [
        fork(
          rule(zero, nat, [t(zero)]),
          [leaf(zero)]
        )
      ]
    ),
    leaf('+'),
    fork(
      rule(const, expr, [n(nat)]),
      [
        fork(
          rule(succ, nat, [t(succ), t('('), n(nat), t(')')]),
          [
            leaf(succ),
            leaf('('),
            fork(
              rule(zero,nat,[t(zero)]),
              [leaf(zero)]
            ),
            leaf(')')
          ]
        )
      ]
    )
  ]
).
```

Online: languages/expr/sample.ptree

BGL parser - top-down

```
% Parse input, non-deterministically and top-down
parseTopDown(
    grammar(_, Rules), % rules to interpret
    Root, % root nonterminal
    Input, % input string of terminals
    Tree % parse tree
  ) :-
    parseTopDown_(Rules, n(Root), Tree, Input, []).

% Consume terminal at top of stack from input
parseTopDown_(_, t(T), leaf(T), [T|Input], Input).

% Expand nonterminal at top of stack
parseTopDown_(Rules, n(N), fork(Rule,Trees1), Input0, Input1) :-
  member(Rule, Rules),
  Rule = rule(_, N, Rhs),
  seq(parseTopDown_(Rules), Rhs, Trees1, Input0, Input1).
```

Online: languages/bgl

Some higher-order bits

```
% EBNF-like sequential composition for accumulating predicates
seq(_, [], X, X).
seq(P, [H|T], X, Z) :-
  apply(P, [H, X, Y]),
  seq(P, T, Y, Z).

% Extension of seq/4 to add list construction
seq(_, [], [], X, X).
seq(P, [H1|T1], [H2|T2], X, Z) :-
  apply(P, [H1, H2, X, Y]),
  seq(P, T1, T2, Y, Z).
```

Online: prelude/higher-order.pro

Let's implode!

```
add(
    const(zero),
    const(succ(zero))
)
```

Implosion

```
% Implosion
implode(
    ETree, % Exploded (detailed) parse tree
    ITree % Imploded (concise) parse tree
  ) :-
      implode_([ETree], [ITree]).

% Base case; implosion completed
implode_([], []).

% Omit terminal from exploded tree in imploded one
implode_(
    [leaf(_)|ETrees], % terminal tree in front
    ITrees % recursively imploded trees
  ) :-
      implode_(ETrees, ITrees).

% Implode subtree recursively
implode_(
    [fork(rule(L, _, _), ETrees1)|ETrees2], % nonterminal tree in front
    [ITree|ITrees2] % binarily recursively imploded trees
  ) :-
      implode_(ETrees1, ITrees1),
      ITree =.. [L|ITrees1], % leverage label as function symbol
      implode_(ETrees2, ITrees2).
```

Online: languages/bgl

Explosion

```
% Explosion
explode(
    grammar(_, Rules), % Rules to consult for details
    Root, % Assumed root nonterminal
    ITree, % Imploded (concise) parse tree
    ETree % Exploded (detailed) parse tree
  ) :-
       explode_(Rules, [n(Root)], [ITree], [ETree]).

% Base case; explosion completed
explode_(_, [], [], []).

% Add heading terminal from rule back into exploded form
explode_(
    Rules,
    [t(T)|Symbols],
    ITrees,
    [leaf(T)|ETrees]
  ) :-
       explode_(Rules, Symbols, ITrees, ETrees).

% Find a rule for the function symbol at hand
explode_(
    Rules,
    [n(N)|Symbols],
    [ITree|ITrees1],
    [fork(Rule, ETrees1)|ETrees2]
  ) :-
       ITree =.. [L|ITrees2],
       Rule = rule(L, N, Rhs),
       member(Rule, Rules),
       explode_(Rules, Rhs, ITrees2, ETrees1),
       explode_(Rules, Symbols, ITrees1, ETrees2).
```

Online: languages/bgl

Let's derive!

```
[zero] nat : 'zero' ;
[succ] nat : 'succ' '(' nat ')' ;
[const] expr : nat ;
[add] expr : expr '+' expr ;
```

```
signature(
  [nat, expr],
  [ symbol(zero, [], nat), % zero ("0")
    symbol(succ, [nat], nat),
    symbol(const, [nat], expr),
    symbol(add, [expr, expr], expr)
  ]
).
```

BGL-to-signature conversion

```
% Convert a grammar to a signature
bglToSignature(
    grammar(Nonterminals, Rules),
    signature(Sorts, STypes)
  ) :-
       Nonterminals = Sorts,
       map(rule2sType, Rules, STypes).

% Convert a rule to a symbol type
rule2sType(
    rule(Label, Lhs, Rhs),
    symbol(Symbol, Arguments, Result)
  ) :-
       Label = Symbol,
       Lhs = Result,
       rhs2arguments(Rhs, Arguments).

% Empty (remaining) RHS maps to null arguments
rhs2arguments([], []).

% Terminals are not mapped to the signature
rhs2arguments([t(_)|Symbols], Sorts) :-
  rhs2arguments(Symbols, Sorts).

% Nonterminals are mapped to sorts
rhs2arguments([n(Nonterminal)|Symbols], [Sort|Sorts]) :-
  Nonterminal = Sort,
  rhs2arguments(Symbols, Sorts).
```

Online: languages/bgl

Isn’t it hard to keep track of all the artifacts and data flows for even just pretty printing and parsing?

We need megamodels!

Topics in this tutorial

- Representation formats
- Basic modeling tasks
- Models of computation
- Pretty printing
- Parsing text to trees
- **Megamodeling (UEBERmodeling)**
- Software transformations
- Reference resolution
- (Structure editing)
- The software ontology SoLaSoTe

Megamodeling

Ralf Lämmel
Software Language Engineer, University of Koblenz-Landau

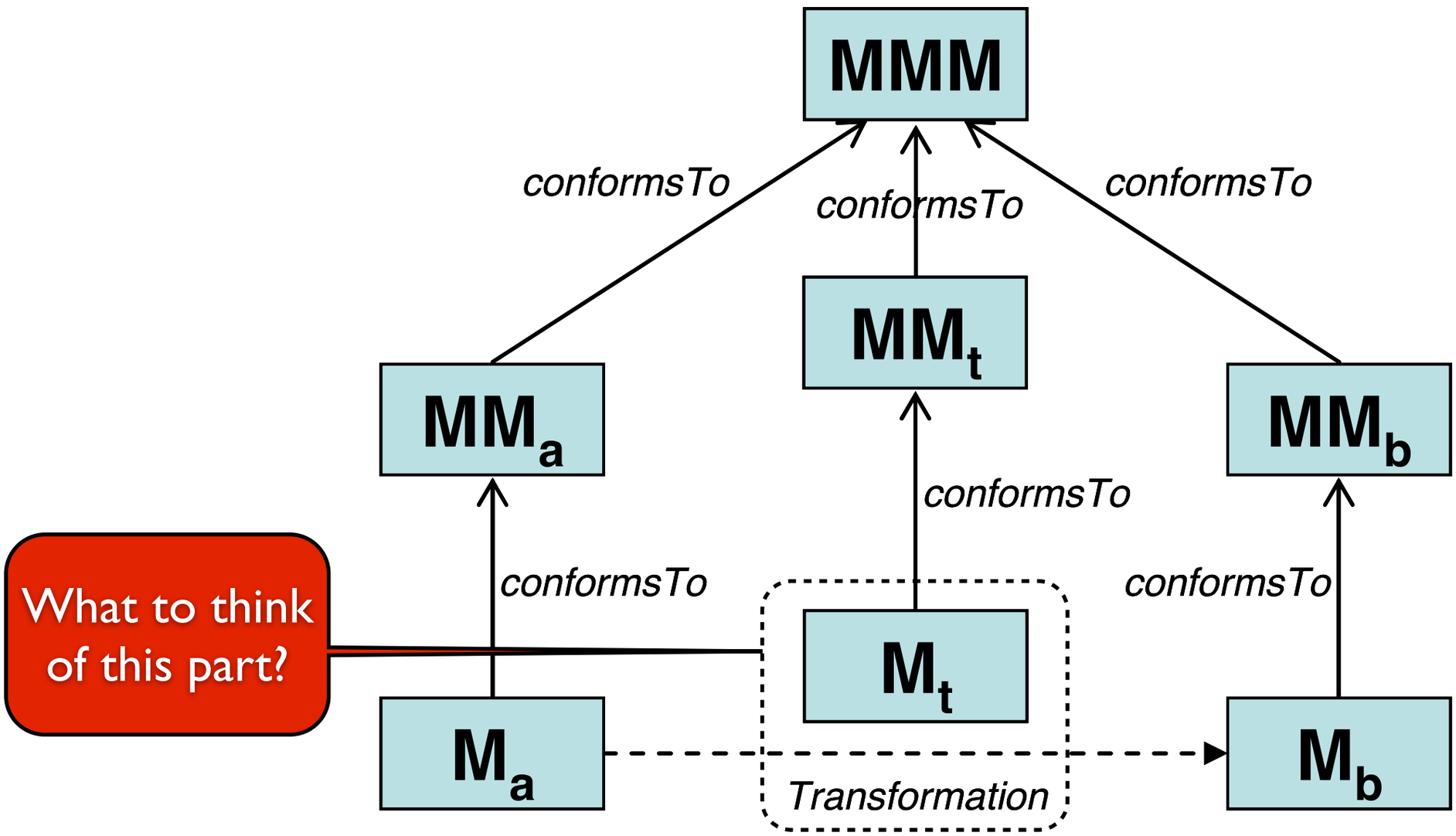
Part of the MODELS'14 tutorial on
"Language Modeling Principles"
http://softlang.uni-koblenz.de/models14/

What's a megamodel?

„A megamodel is a model of which at least some elements represent and/or refer to models or metamodels.“ [Bezivin, Jouault, Valduriez; 2004]

About everything is a model: data, programs, metamodels, model transformations, …

A megamodel for ATL's MT mechanics

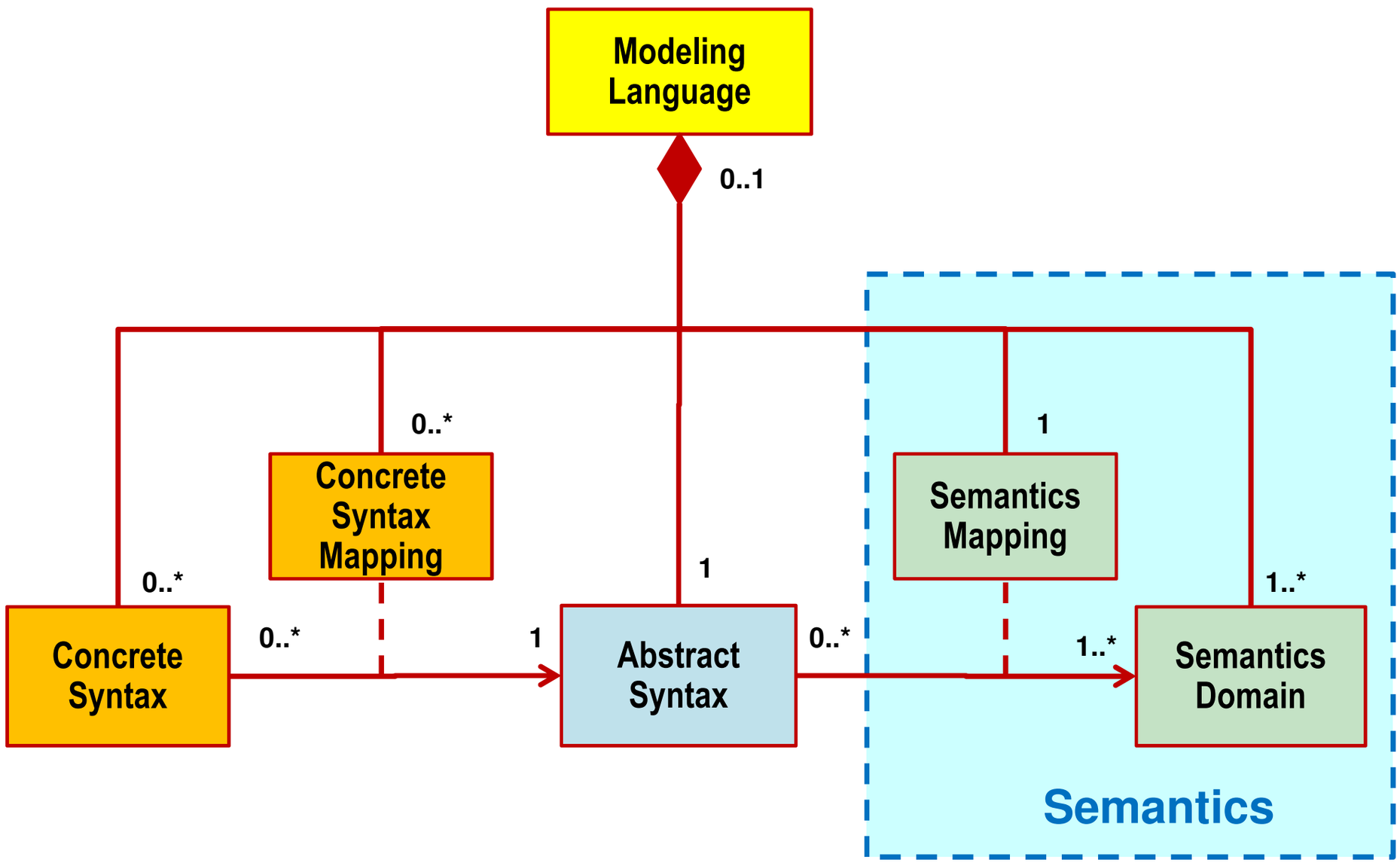


Source:
http://wiki.eclipse.org/ATL/Concepts#Model_Transformation

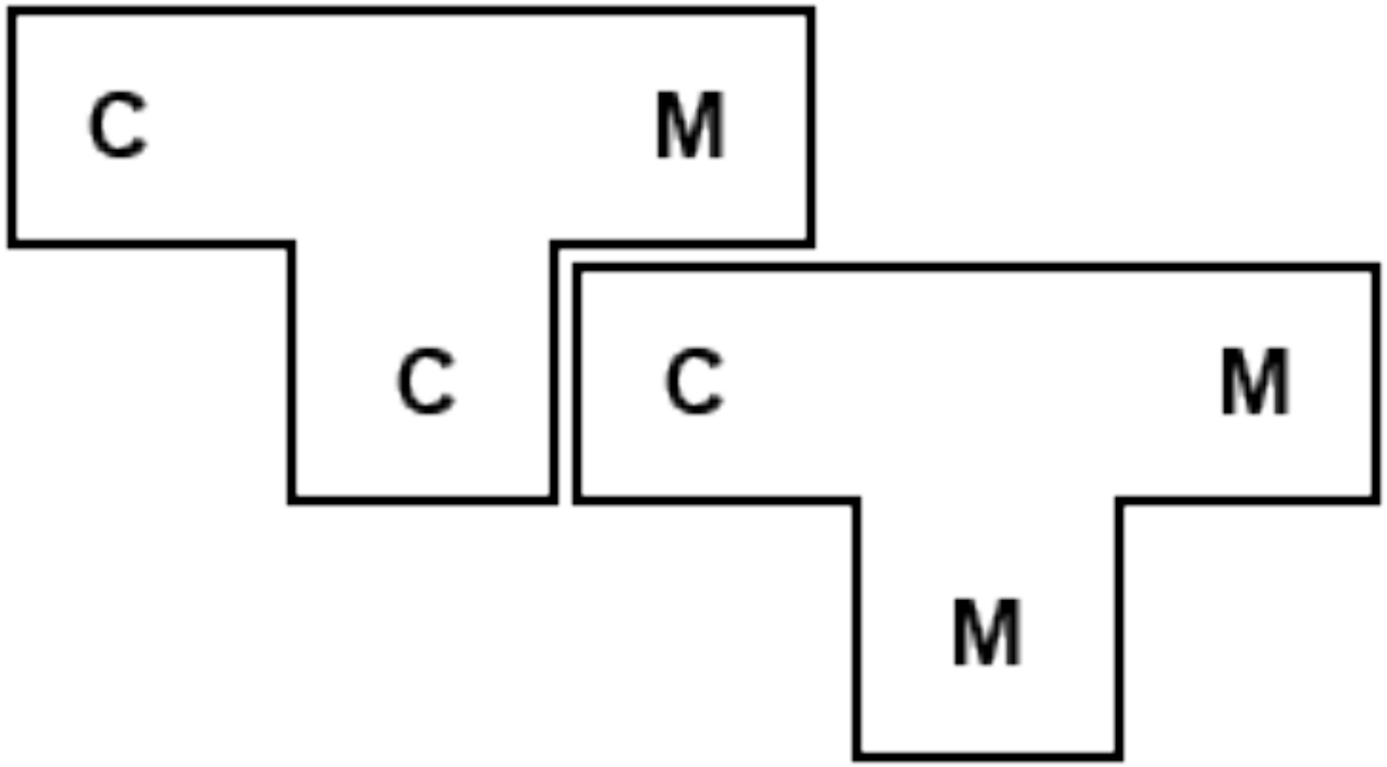
Elements of a Modeling Language

Source: Bran Selic: “The “Theory” and Practice of Modeling Language Design, Tutorial at MODELS’13.

Selic’s model can be interpreted as a megamodel.

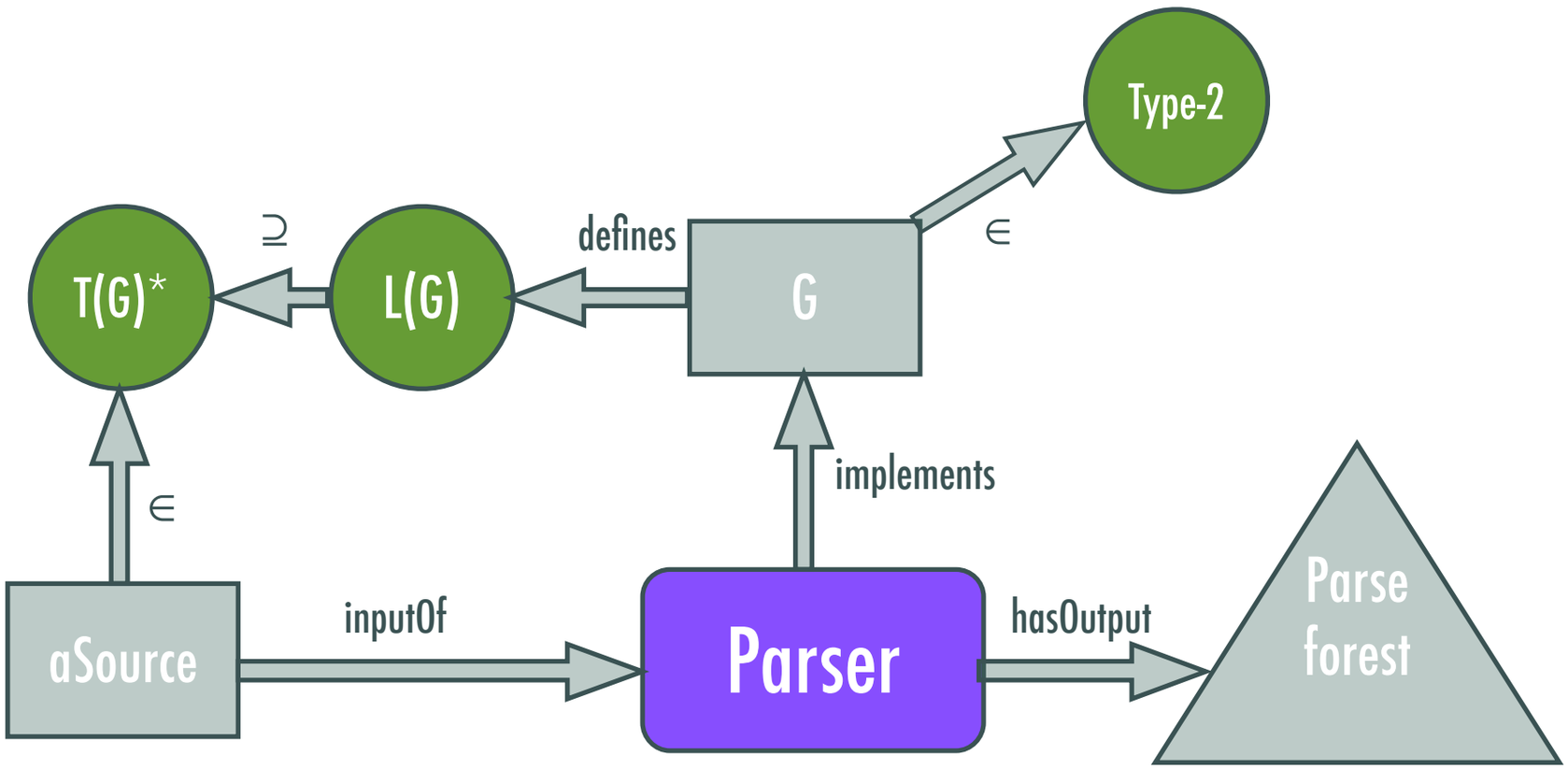


A classic „megamodel“ for bootstrapping a compiler

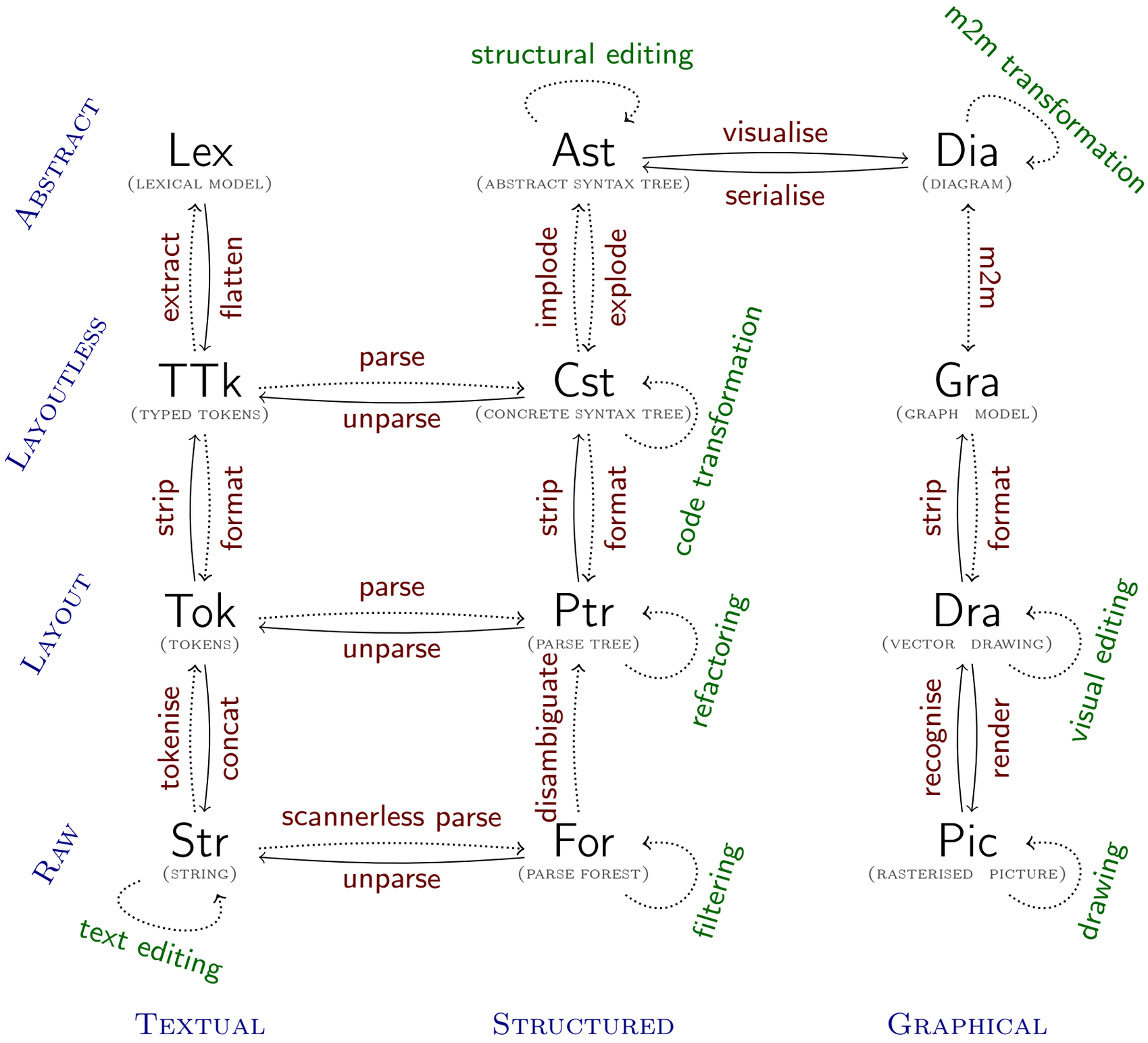


http://en.wikipedia.org/wiki/Tombstone_diagram

A megamodel to explain parsing



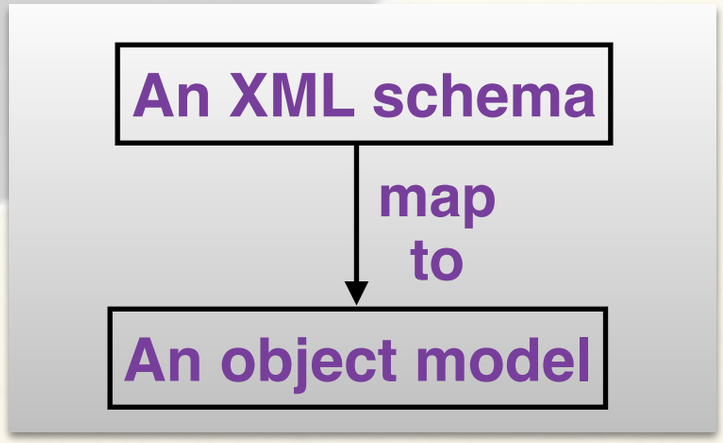
A megamodel to explain parsing et al.



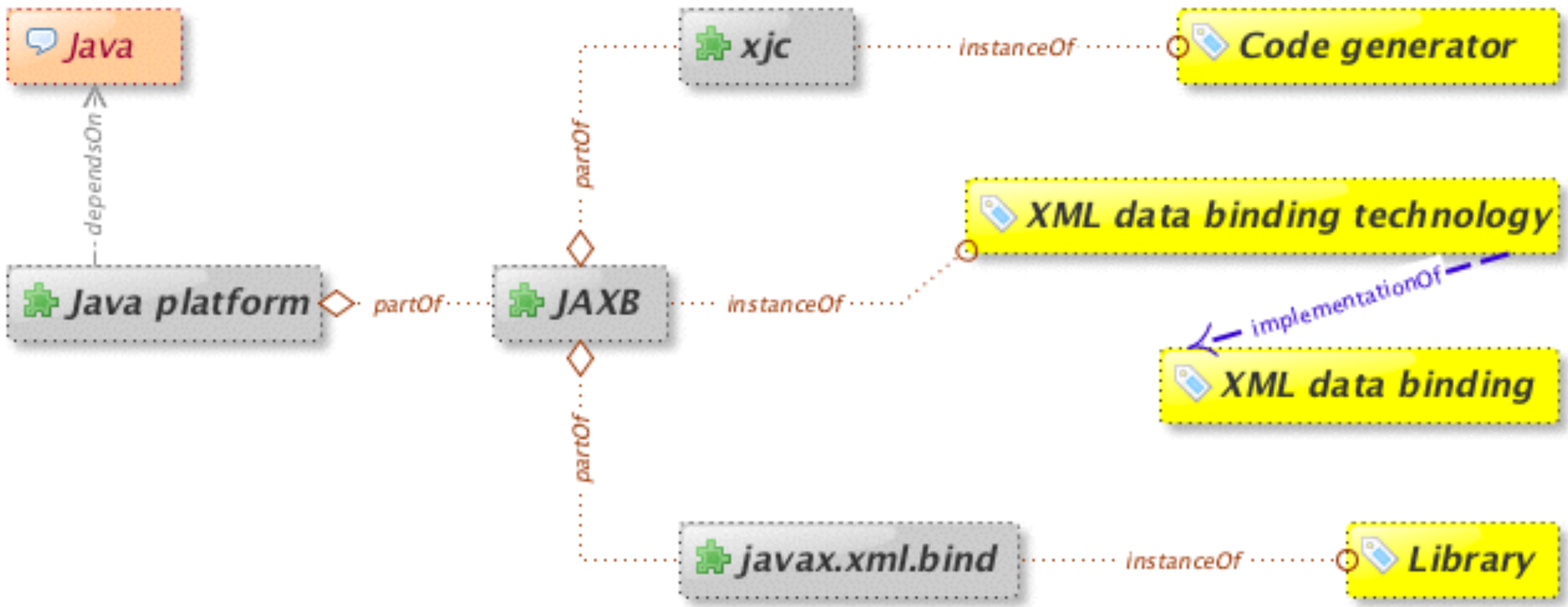
[BaggeZ14]
(MODELS'14)

Technology models are megamodels

- Identify constituents of software technologies
- Identify artifacts involved by technology usage
- Identify languages for the involved artifacts
- Identify data flows representing technology usage
- Identify concepts implied by technology usage
- …



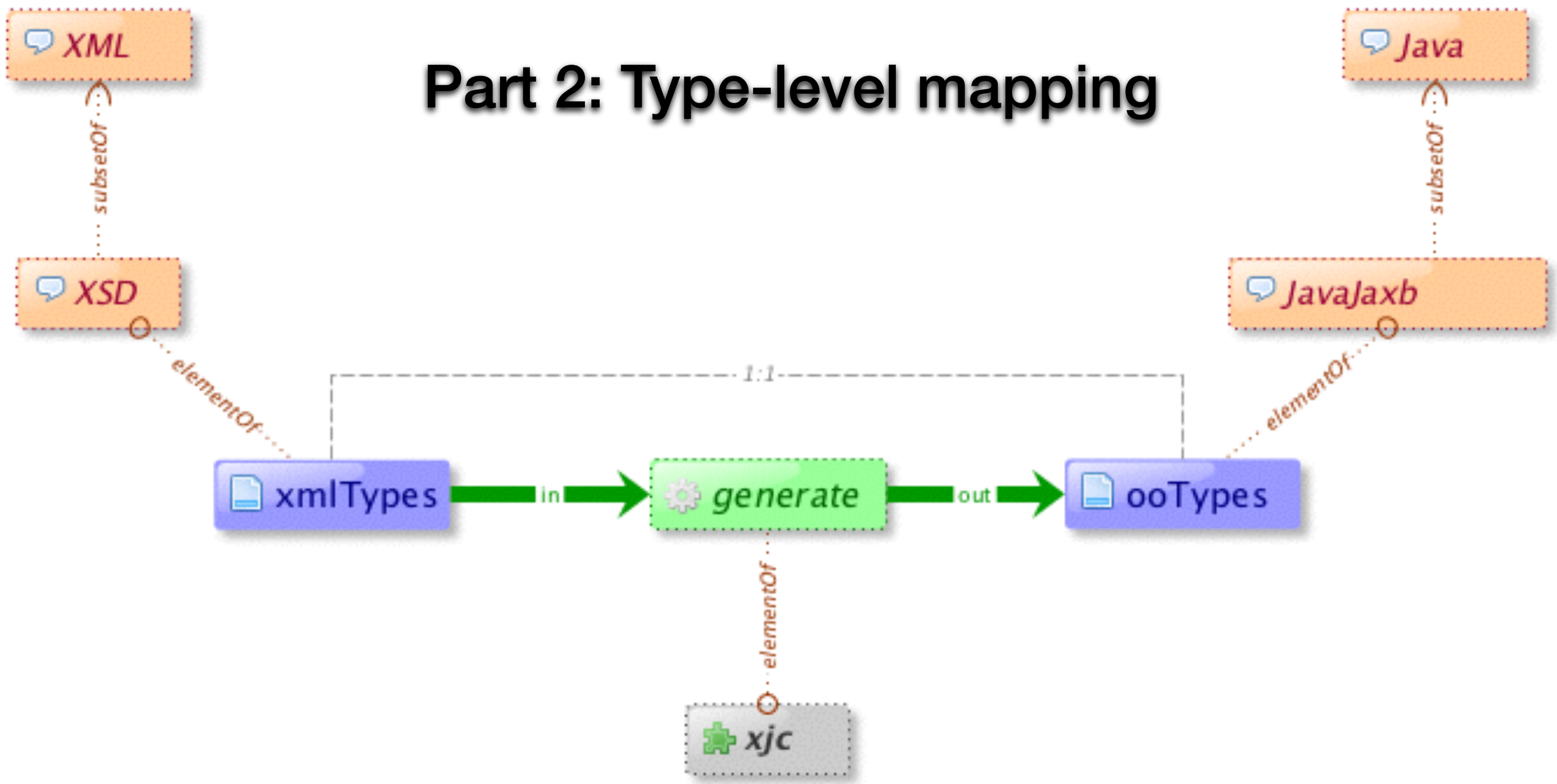
A megamodel for the JAXB technology (XML-data binding of the Java platform)



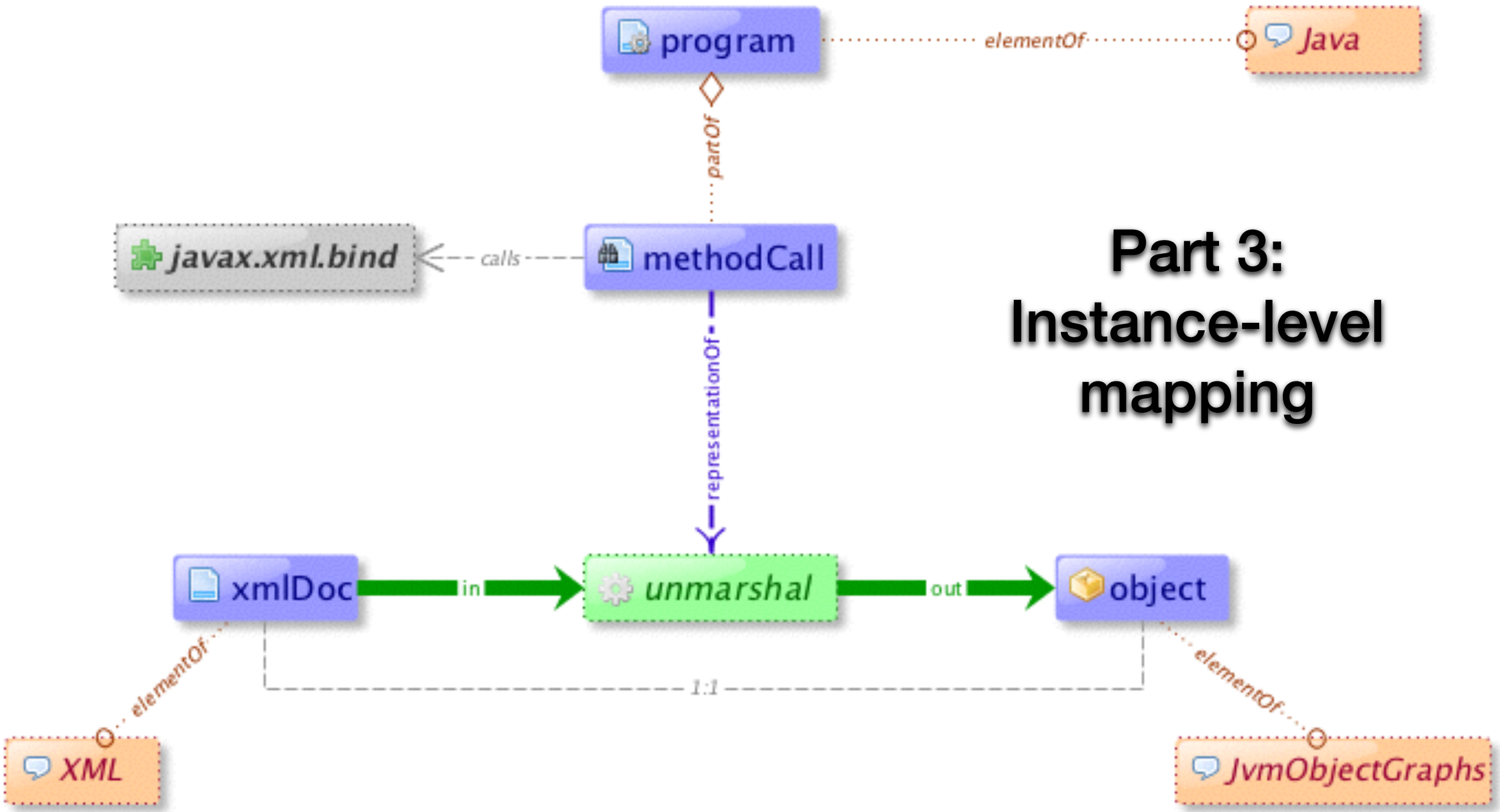
Part 1: Technology break-down and concepts

A megamodel for the JAXB technology (XML-data binding of the Java platform)

Part 2: Type-level mapping

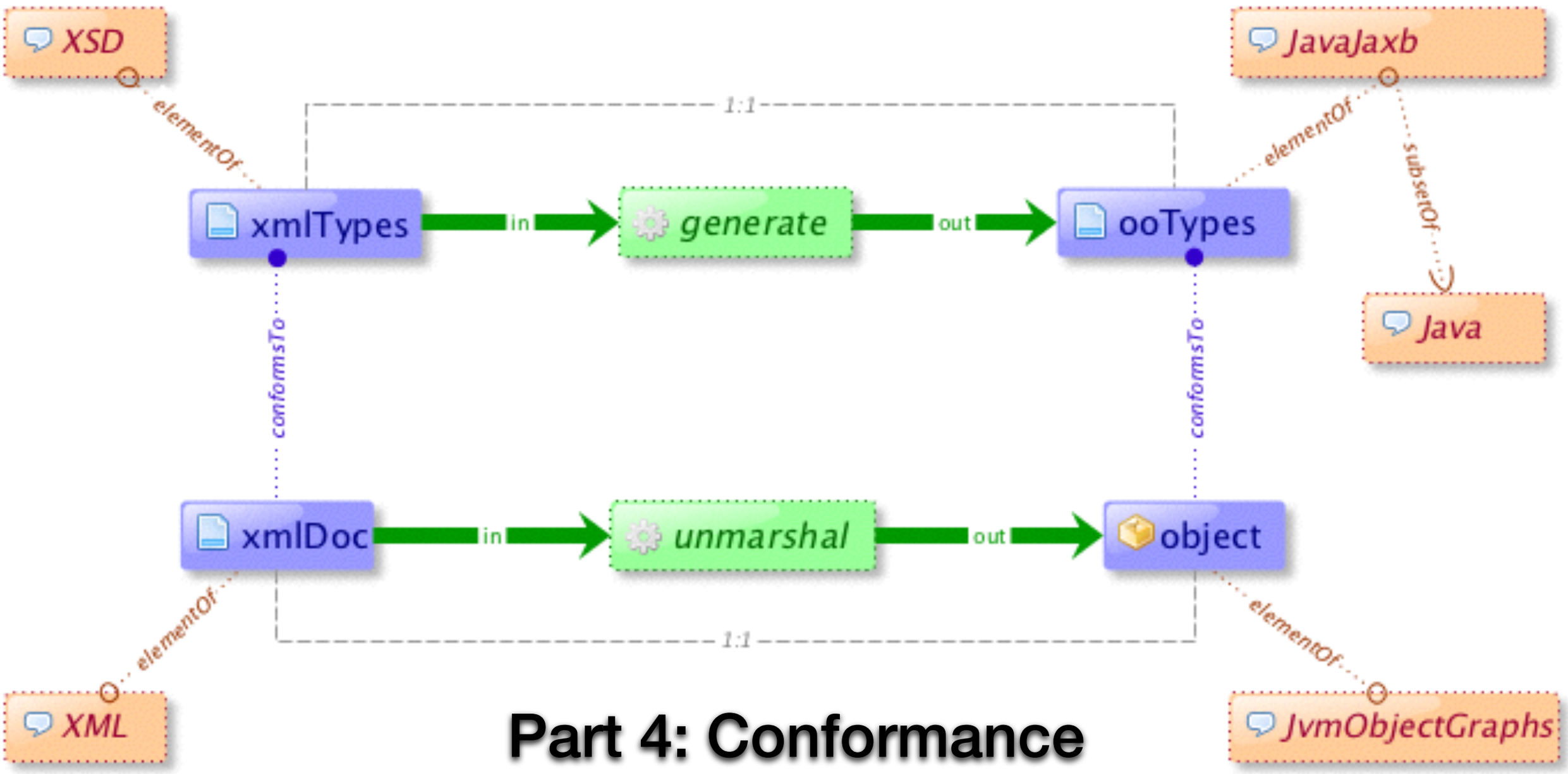


A megamodel for the JAXB technology (XML-data binding of the Java platform)



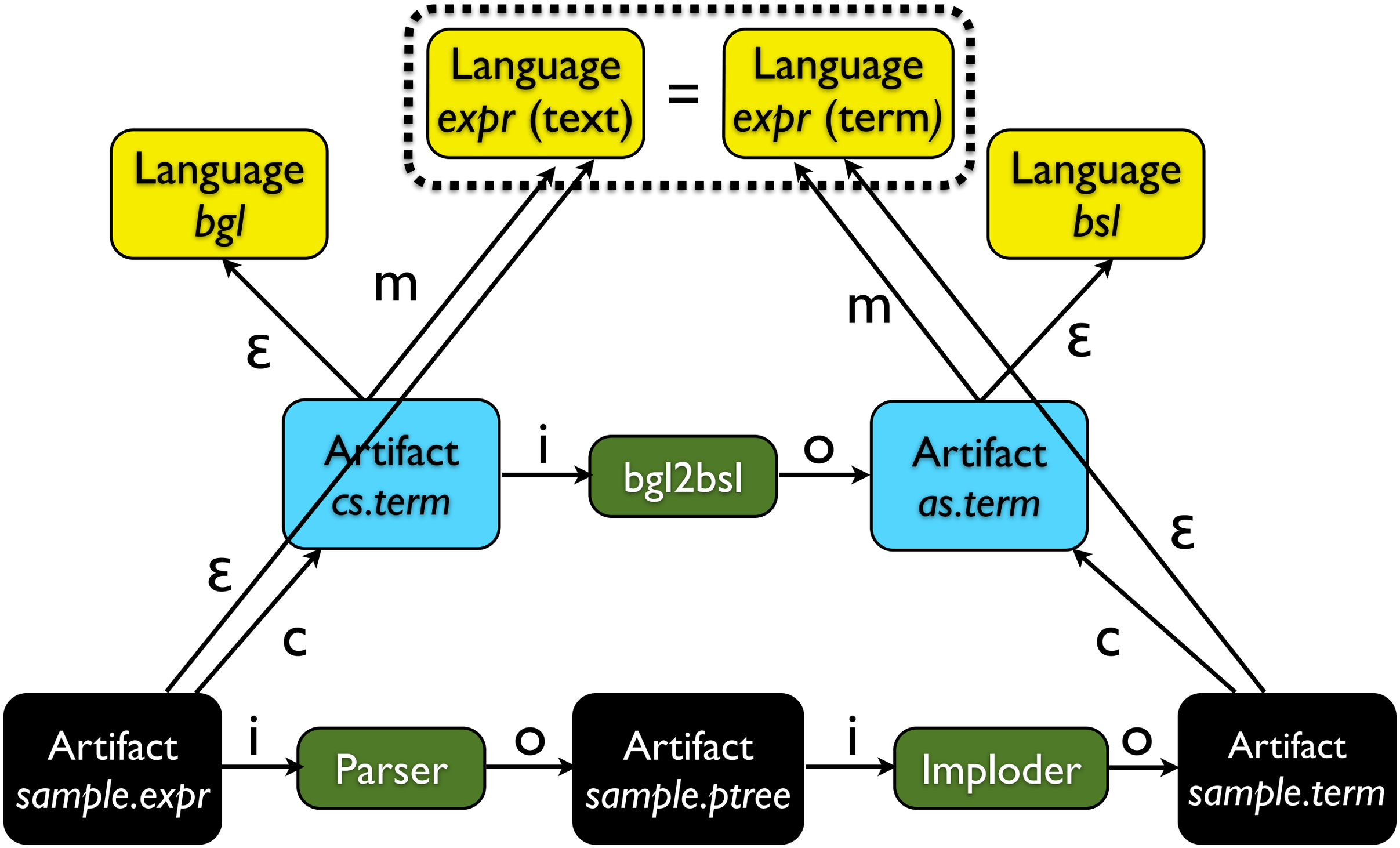
Part 3: Instance-level mapping

A megamodel for the JAXB technology (XML-data binding of the Java platform)



Part 4: Conformance

A megamodel for some *EXPR* components



The megamodeling language *UEBER*

Central concepts

- Languages and membership
- Functions and application
- Language-typed artifacts with data flow

Important characteristics

- Executable megamodeling language
- Replacement for scripting and testing
- Homogenous situation („Everything is Prolog.“)

An *UEBER* model of *EXPR*

```
[

  % Manifestations of the EXPR language
  language(expr(text)), % Text notation
  language(expr(tokens(term))), % Tokenized text
  language(expr(ptree(term))), % Raw parse trees
  language(expr(term)), % Imploded parse trees

  ...
```

Online: languages/expr/.ueber

Syntax of *UEBER*

```
type ueber = udecl* ;
symbol language : lang -> udecl ;
symbol membership : lang x goal x file* -> udecl ;
symbol equivalence : lang x goal x file* -> udecl ;
symbol function : func x lang+ x lang+ x goal x file* -> udecl ;
symbol elementOf : file x lang -> udecl ;
symbol mapsTo : func x file+ x file+ -> udecl ;
symbol macro : goal -> udecl ;
type file = atom ;
type func = atom ;
type lang = term ;
type goal = term ;
```

Online: languages/ueber/as.esl

An *UEBER* macro for **parsing**

```
% Reusable pattern of parsing a sample
parse(TextFile) :-
  name(TextFile, Str),
  append(StemStr, [0'.|LangStr], Str),
  name(Lang, LangStr),
  name(Stem, StemStr),
  TextLang =.. [Lang, text],
  TermLang =.. [Lang, term],
  atom_concat([Stem, '.term'], TermFile),
  declare(elementOf(TextFile, TextLang)),
  declare(elementOf(TermFile, TermLang)),
  declare(mapsTo(parser, [TextFile], [TermFile])).
```

Online: languages/ueber/macros

An *UEBER* macro for **syntax definitions**

% Reusable pattern of concrete and abstract syntax definition

```
syntax(Lang) :-
  TextLang =.. [Lang, text],
  TermLang =.. [Lang, term],
  declare(language(TextLang)),
  declare(language(TermLang)),
  ConSyn = ['cs.term', 'ls.term'],
  AbsSyn = ['as.term'],
  atom_concat(Lang, 'Mapping', Mapping),
  declare(membership(TextLang, eglAcceptor(Mapping), ConSyn)),
  declare(membership(TermLang, eslChecker, AbsSyn)),
  declare(function(parser, [TextLang], [TermLang], eglParser(Mapping), ConSyn)),
  declare(elementOf('cs.egl', egl(text))),
  declare(elementOf('cs.term', egl(term))),
  declare(elementOf('ls.egl', egl(text))),
  declare(elementOf('ls.term', egl(term))),
  declare(elementOf('as.esl', esl(text))),
  declare(elementOf('as.term', esl(term))),
  declare(mapsTo(parser, ['cs.egl'], ['cs.term'])),
  declare(mapsTo(parser, ['ls.egl'], ['ls.term'])),
  declare(mapsTo(parser, ['as.esl'], ['as.term'])).
```

Online: languages/ueber/macros

An *UEBER* model for *FSML*

```
[
  macro(syntax(fsml)),
  macro(parse('sample.fsml')),
  language(fsml(ok(term))),
  membership(fsml(ok(term)), okFsm, []),
  ...
```

Online: languages/fsml/.ueber

Java code concerns of FSML

```
[
  elementOf('State.java', java(text)),
  elementOf('Input.java', java(text)),
  elementOf('Action.java', java(text)),
  elementOf('Handler.java', java(text)),
  elementOf('Stepper.java', java(text)),
  elementOf('Demo.java', java(text)),
  elementOf('HandlerBase.java', java(text)),
  elementOf('StepperBase.java', java(text)),
  elementOf('Pair.java', java(text))
].
```

Online: languages/fsml/java/.ueber

Test suite for FSML

```
[
  not(elementOf('parserError.fsml', fsml(text))),
  macro(parse('initialNotOk.fsml')),
  not(elementOf('initialNotOk.term', fsml(ok(term)))),
  macro(parse('idsNotOk.fsml')),
  not(elementOf('idsNotOk.term', fsml(ok(term)))),
  macro(parse('resolutionNotOk.fsml')),
  not(elementOf('resolutionNotOk.term', fsml(ok(term)))),
  macro(parse('determinismNotOk.fsml')),
  not(elementOf('determinismNotOk.term', fsml(ok(term)))),
  macro(parse('reachabilityNotOk.fsml')),
  not(elementOf('reachabilityNotOk.term', fsml(ok(term)))),
  elementOf('illegalSymbol.input', term),
  elementOf('infeasibleSymbol.input', term),
  not(mapsTo(acceptFsm, ['../sample.term', 'illegalSymbol.input'], [])),
  not(mapsTo(acceptFsm, ['../sample.term', 'infeasibleSymbol.input'], []))
].
```

Online: languages/fsml/tests/.ueber

Topics in this tutorial

- Representation formats
- Basic modeling tasks
- Models of computation
- Pretty printing
- Parsing text to trees
- Megamodeling (UEBERmodeling)
- **Software transformations**
- Reference resolution
- (Structure editing)
- The software ontology SoLaSoTe

Exogenous software transformations

Ralf Lämmel
Software Language Engineer, University of Koblenz-Landau

Part of the MODELS'14 tutorial on
"Language Modeling Principles"
http://softlang.uni-koblenz.de/models14/

Transformation development

- Pick the (abstract) syntax of input.
- Pick the (abstract) syntax of output.
- Set up test case(s).
- Rewrite input to output.
- Validate transformation with test case(s).

Metamodel of the *family* language

```
metamodel family {

  class family {
    value name : atom;
    part members : person*;
  }

  class person {
    value firstName : atom;
    value emailAddresses : atom*;
    reference closestFriend : person?;
  }

  datatype atom;

}
```

Online: languages/family/mm2.mml

Relational schema for *family*

```
CREATE TABLE family (
  objectId INTEGER NOT NULL PRIMARY KEY,
  name VARCHAR(42) NOT NULL
);

CREATE TABLE person (
  objectId INTEGER NOT NULL PRIMARY KEY,
  firstName VARCHAR(42) NOT NULL,
  closestFriend INTEGER FOREIGN KEY REFERENCES person (objectId)
);

CREATE TABLE family_members (
  familyId INTEGER NOT NULL FOREIGN KEY REFERENCES family (objectId),
  members INTEGER NOT NULL FOREIGN KEY REFERENCES person (objectId)
);

CREATE TABLE person_emailAddresses (
  personId INTEGER NOT NULL FOREIGN KEY REFERENCES person (objectId),
  emailAddresses VARCHAR(42) NOT NULL
);
```

Online:
languages/family/dd.sql

Abstract syntax of metamodels

```
alias(metamodel, tuple([
    sort(name),
    list(sort(classifier))
  ])),

symbol(class, [
    sort(abstract),
    sort(name),
    option(sort(extends)),
    list(sort(member))
  ], classifier),

...
```

Excerpt

Online: languages/mml

Abstract syntax of DDL subset

```
alias(schema, list(sort(table))),

alias(table, tuple([
    sort(name),
    list(sort(column))
  ])),

alias(column, tuple([
    sort(name),
    sort(type),
    list(sort(clause))
  ])),

...
```

Online: languages/ddl

Simplicity ("mappability")

```
% Simplicity of metamodels
simpleMetamodel((_, Classifiers)) :-
  map(simpleClassifier, Classifiers).

% Simplicity of classes
simpleClassifier(
  class(
    false, % Concrete classes, only
    Name,
    [], % Classes without super, only
    _ % No constraints on members
  ))
 :-
    \+ datatype(Name).

% Simplicity of datatypes
simpleClassifier(datatype(X)) :-
  datatype(X).

% All known datatypes
datatype(atom).
datatype(integer).
```

Online:
languages/mml/to-ddl

Map classes to tables 1/4

```
% Map classes to tables
classesToTables((_, Classifiers), Tables3) :-
  map(classToTable, Classifiers, Tablesss1),
  concat(Tablesss1, Tables1),
  map(multisToTables, Classifiers, Tablesss2),
  concat(Tablesss2, Tables2),
  append(Tables1, Tables2, Tables3).
```

Online: languages/mml/to-ddl

Map classes to tables 2/4

```
% Map each class to a table
classToTable(

  class(
    false, % Map concrete classes, only
    Name,
    [], % Map non-extended classes, only
    Members),

 [ (Name,
    [ PrimaryKey % Standard column for primary key
    | Columns % Columns for single-valued members
    ] ) ] )

:-
    ...
```

Excerpt

Online: languages/mml/to-ddl

Map classes to tables 3/4

```
% Map value members
singleToColumn(
  (value, Name, Type, Cardinality),
  [(Name, SqlType, Clauses)])
 :-
    singleCardinality(Cardinality, Clauses),
    datatypeToSql(Type, SqlType).

% Map non-value members
singleToColumn(
  (Kind, Name, Type, Cardinality),
  [(Name, integer, Clauses2)])
 :-
    member(Kind, [part, reference]),
    singleCardinality(Cardinality, Clauses1),
    append(Clauses1, [foreignKey(Type, objectId)], Clauses2).
```

Online: languages/mml/to-ddl

Map classes to tables 4/4

```
% Map multi-valued member to a designated table
multiToTable(
  Class, % Class containing the multi-valued member
  (Kind, Member, Type, star), % The multi-valued member
  [(Table, [From, To])]) % The relationship table
 :-
    atom_concat([Class, '_', Member], Table),
    downcase_atom(Class, LowerCase),
    atom_concat([LowerCase, 'Id'], Column),
    From = (
        Column, % Synthesized column name
        integer, % Type for SQL keys
        [notNull, foreignKey(Class, objectId)]
   ),
singleToColumn((Kind, Member, Type, one), [To]).
```

Excerpt

Online: languages/mml/to-ddl

Reflections

- We would also need instance mapping.
- We could also operate on graphs.
- Concrete object syntax may be desirable.
- Different schemes may be applied to inheritance.

Endogenous software transformations

Ralf Lämmel
Software Language Engineer, University of Koblenz-Landau

Omitted for

Part of the MODELS'14 tutorial on
"Language Modeling Principles"
http://softlang.uni-koblenz.de/models14/

Topics in this tutorial

- Representation formats
- Basic modeling tasks
- Models of computation
- Pretty printing
- Parsing text to trees
- Megamodeling (UEBERmodeling)
- Software transformations
- **Reference resolution**
- (Structure editing)
- The software ontology SoLaSoTe

Reference resolution

Ralf Lämmel
Software Language Engineer, University of Koblenz-Landau

Part of the MODELS'14 tutorial on
"Language Modeling Principles"
http://softlang.uni-koblenz.de/models14/

Motivation

- Data often contains implicit references:
  - In a program, variable ids refer to declaration.
  - ...
- Implicit references could be made explicit.
  - That is, actual links are added to “model”.
- Such resolution is relevant across paradigm.
  - Let us resolve in a metamodeling context.

Metamodel of the *family* language

```
metamodel family {

  class family {
    value name : atom;
    part members : person*;
  }

  class person {
    value firstName : atom;
    value emailAddresses : atom*;
    value closestFriend : atom?;
 }

  datatype atom;

}
```

This metamodel suffices with tree shape.

Online: languages/family/mm1.mml

Another metamodel

```
metamodel family {

  class family {
    value name : atom;
    part members : person*;
  }

  class person {
    value firstName : atom;
    value emailAddresses : atom*;
    reference closestFriend : person?;
  }

  datatype atom;

}
```

This metamodel involves **graph** shape.

Online: languages/family/mm2.mml

A model which is a **tree**

```
{
  class : family,
  name : smallFamily,
  members : [
    {
      class : person,
      firstName : x,
      emailAddresses : [ 'x@small.family.com', 'x42@earth.com' ],
      closestFriend : [ y ]
    },
    {
      class : person,
      firstName : y,
      emailAddresses : [ ],
      closestFriend : [ ]
    }
  ]
}.
```

Online: languages/family/sample-small-mm1.graph

A model which is a **graph**

```
{
  class : family,
  name : smallFamily,
  members : [
    42 &
    {
      class : person,
      firstName : x,
      emailAddresses : [ 'x@small.family.com', 'x42@earth.com' ],
      closestFriend : [ #88 ]
    },
    88 &
    {
      class : person,
      firstName : y,
      emailAddresses : [ ],
      closestFriend : [ ]
    }
  ]
}.
```

Online: languages/family/sample-small-mm2.graph

Coupled metamodel/model transformation 1/2

```
% Replace an atom by a reference
atomToRef(
    CFrom, % Referring class
    CTo, % Referred class
    KFrom, % Key on referring class
    KTo, % Key on referred class
    M1, % Input model
    MM1, % Input metamodel
    M2, % Output model
    MM2 % Output metamodel
  )

:-
    ...
```

Online: languages/mml/atom-to-ref

Coupled metamodel/model transformation 2/2

```
... :-

    % Precondition(s)
    conforms(M1, MM1),

    % Metamodel level
    atomToRefMM(CFrom, CTo, KFrom, MM1, MM2),

    % Model level
    ZFrom = instanceOf(MM1, CFrom),
    ZTo = instanceOf(MM1, CTo),
    atomToRefM(ZFrom, ZTo, KFrom, KTo, M1, M2),

    % Postcondition(s)
    conforms(M2, MM2).
```

Online: languages/mml/atom-to-ref

Metamodel transformation

```
% atomToRef at metamodel level
atomToRefMM(
  From, % Class that is referring
  To, % Class that is being referred to
  Key, % Key to be updated
  MM1, % Input metamodel
  MM2 % Output metamodel
 ) :-
      require(
        memberMissing(From, Key),
        lookupMember(From, Key, MM1, X1)),
      require(
        valueMissing,
        lookup(class, X1, value)),
      require(
        atomMissing,
        lookup(type, X1, #atom)),
      update(class, reference, X1, X2),
      update(type, #To, X2, X3),
      updateMember(From, X3, MM1, MM2).
```

Online: languages/mml/atom-to-ref

Model transformation

```
% Replace atom-typed reference by actual reference
atomToRefM(
  From, % Referring objects
  To, % Referred objects
  KRef, % Key for reference on "From" objects
  KId, % Corresponding key on "To" objects
  M1, % Input model
  M3 % Output model
 ) :-
      graphNf(M1, M2), % All objects have IDs
      topdownGraph( % Iterate over the object
        atomToRefM_(From, To, KRef, KId, M2),
        M2, M3).

...
```

Online: languages/mml/atom-to-ref

Excerpt

Reflections

- Other reference resolution schemes are needed.
  - Data other than an atom-typed value.
  - Selection other than local object inspection.
- Coupled transformation is an active research area.
- Reference resolution needed across paradigm.
  - Environments in functional programming
  - References for “extended” attribute grammars
  - ...

Topics in this tutorial

- Representation formats
- Basic modeling tasks
- Models of computation
- Pretty printing
- Parsing text to trees
- Megamodeling (UEBERmodeling)
- Software transformations
- Reference resolution
- **(Structure editing)**
- The software ontology SoLaSoTe

Structure editing

Ralf Lämmel
Software Language Engineer, University of Koblenz-Landau

Omitted for

Part of the MODELS'14 tutorial on
"Language Modeling Principles"
http://softlang.uni-koblenz.de/models14/

Topics in this tutorial

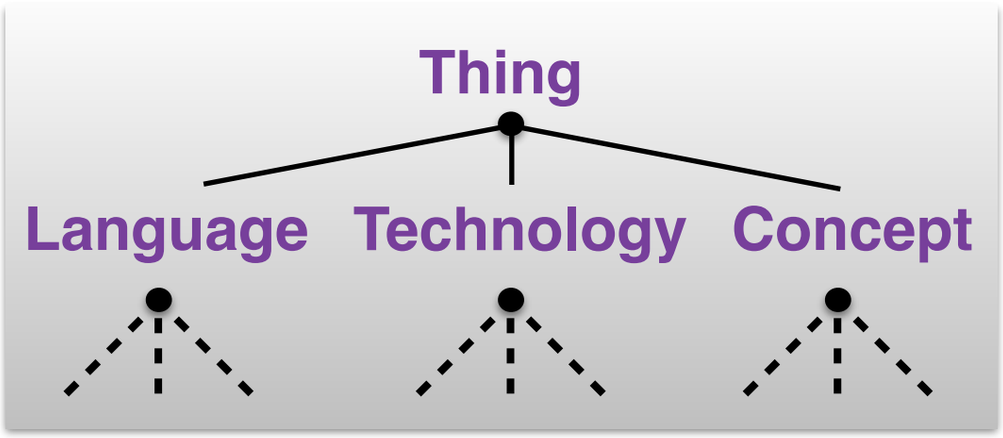
- Representation formats
- Basic modeling tasks
- Models of computation
- Pretty printing
- Parsing text to trees
- Megamodeling (UEBERmodeling)
- Software transformations
- Reference resolution
- (Structure editing)
- **The software ontology SoLaSoTe**

The software ontology *SoLaSoTe*

Ralf Lämmel
Software Language Engineer, University of Koblenz-Landau

Part of the MODELS'14 tutorial on
"Language Modeling Principles"
http://softlang.uni-koblenz.de/models14/

Classification with SoLaSoTe



Types of *SoLaSoTe*'s individuals

type	comment
onto:Concept	Software concepts
onto:Contribution	Contributions to the 101 project
onto:Contributor	Contributors to the 101 project
onto:Course	Courses on programming and software engineering
onto:Document	Documents in a broad sense
onto:Feature	Software features
onto:Language	Software languages
onto:Script	Scripts as units of a course
onto:Technology	Software technologies
onto:Theme	Containers of contributions
onto:Vocabulary	Containers of terms

For instance: software **languages**

language	headline
Java	An OO programming language
Haskell	A purely-functional programming language
XML	The extensible markup language
JavaScript	A multi-paradigm programming language for the web et al.
JSON	The JavaScript Object Notation for data exchange
SQL	Data definition and manipulation for relational databases
Python	A multi-paradigm programming language
...	

For instance: software **technologies**

technology	headline
Gradle	A build tool inspired by Ant and Maven
JUnit	A framework for unit testing for Java
Eclipse	An IDE for Java with a plug-in system
.NET	A library and runtime for programming languages on Windows
ANTLR	A parser generator with various language processing capabilities
GHC	A Haskell compiler
MySQL	A relational database management system
...	

For instance: software **concepts**

concept	headline
Web_programming	The domain of web application development
Algebraic_data_type	A type for the construction of terms
OO_programming	The object-oriented programming paradigm
Functional_programming	The functional programming paradigm
API	An interface for reusable functionality
Type_class	An abstraction mechanism for polymorphism
Software_system	A system of intercommunicating software components
...	

A SPARQL query
sorting software concepts by popularity

```
SELECT ?concept ?headline (COUNT(?subject) AS ?count)
WHERE {
  ?concept a onto:Concept .
  ?concept onto:hasHeadline ?headline .
  ?subject ?predicate ?concept .
}
GROUP BY ?concept ?headline
ORDER BY DESC(?count)
```

Thus, the realization of *SoLaSoTe* depends on RDF, RDFS (OWL), and SPARQL.

SoLaToSe aspects

- **Classification (instanceOf, isA)**
- **Relationships (uses, supports, …)**
- Containers (themes, vocabularies, …)
- Systems (101)
- Other resources (sameAs, …)
- **Validation of the ontology**

Classification

```
ASK {
  lang:Java a onto:Language
}
```

Is ‚Java' a language?

true

```
SELECT ?type
WHERE {
  lang:Java rdf:type ?type .
  ?type a onto:Classifier
}
ORDER BY ?type
```

What are the supertypes of ‚Java'?

```
onto:OO_programming_language
onto:Programming_language
```

```
SELECT ?concept ?headline (COUNT(?subject) AS ?count)
WHERE {
  ?concept a onto:Concept .
  ?concept onto:hasHeadline ?headline .
  ?classifier a onto:Classifier .
  ?classifier onto:classifies ?concept .
  ?subject ?predicate ?concept
}
GROUP BY ?concept ?headline
ORDER BY DESC(?count)
```

What are popular classifiers?

concept
Algebraic_data_type
OO_programming
Functional_programming
API
Software_system
Client
Web_browser
...

Predicates for relationships

predicate	comment
basedOn	Reuse of systems
carries	Tagging of entities
dependsOn	Dependence
designedBy	Designer of a
developedBy	Developer of
illustrates	Chrestomath
implements	Systems im ementing descriptions
linksTo	Non-spe ic link to external resource
memberOf	Mem ership
mentions	Nonspecific r
moreComplexThan	Comparison
partOf	
profile	Web page wi
reviewedBy	Reviewer of an entity
sameAs	Equivalence
similarTo	Similarity rel
supports	
uses	Use of instru
varies	Similarity of

For example:
feature:**Hierarchical**_company
moreComplexThan
feature:**Flat**_company

For example:
tech:**ghc**
partOf
tech:**Haskell_platform**

For example:
tech:**Ruby_on_Rails**
supports
concept:**REST**

Predicates for relationships

predicate	domain	range
onto:basedOn	onto:System	onto:System
onto:carries	onto:Entity	onto:Tag
onto:dependsOn	onto:Entity	onto:Entity
onto:designedBy	onto:Entity	foaf:Person
onto:developedBy	onto:Entity	foaf:Person
onto:illustrates	onto:Description	onto:Instrument
onto:implements	onto:System	onto:Description
onto:linksTo	onto:Entity	rdfs:Literal
onto:memberOf	onto:Entity	onto:Container
onto:mentions	onto:Entity	onto:Entity
onto:moreComplexThan	onto:Entity	onto:Entity
onto:partOf	onto:Entity	onto:Entity
onto:profile	onto:Contributor	rdfs:Literal
onto:reviewedBy	onto:Entity	foaf:Person
onto:sameAs	onto:Entity	rdfs:Literal
onto:similarTo	onto:Entity	rdfs:Literal
onto:supports	onto:Instrument	onto:Instrument
onto:uses	onto:System	onto:Instrument
onto:varies	onto:System	onto:System

Query: *Arrange lecture in a course*

```
SELECT DISTINCT
    ?course
    (COUNT(?prerequisites) AS ?count)
WHERE {
    ?course onto:memberOf course:Lambdas_in_Koblenz .
    OPTIONAL { ?course onto:dependsOn+ ?prerequisites }
}
GROUP BY ?course
ORDER BY ?count
```

course	count
First_steps_in_Haskell	0
Basic_software_engineering_for_Haskell	1
Searching_and_sorting_in_Haskell	2
Basic_data_modeling_in_Haskell	3
Higher-order_functions_in_Haskell	4
Type-class_polymorphism	5
Functional_data_structures	5
Functors_and_friends	6
Generic_functions	7
Unparsing_and_parsing_in_Haskell	7
Monads	8

Arrange scripts (lectures) in an order that respect the lecture dependencies.

RDFS' inference vs. *SPARQL's* validation

RDFS

```
onto:contribUsesLang
  rdfs:type rdfs:Property ;
  rdfs:subPropertyOf onto:uses ;
  rdfs:comment "Use of languages by contributions" ;
  rdfs:domain onto:Contribution;
  rdfs:range onto:Language .
```

Semantics: if a resource is the subject of ‚contribUsesLang', then it is of type `Contribution'.

SPARQL

```
SELECT ?x {
  ?x onto:contribUsesLang ?y .
  FILTER NOT EXISTS { ?x sesame:directType onto:Contribution }
}
```

Semantics: find resources that are subjects of ‚contribUsesLang' without being of declared type `Contribution'.

OWL's consistency vs. *SPARQL's* validation

OWL

```
<owl:AllDisjointClasses>
<owl:members rdf:parseType="Collection">
  <owl:class rdf:about="http://101companies.org/ontology#Language"/>
  <owl:class rdf:about="http://101companies.org/ontology#Technology"/>
  <owl:class rdf:about="http://101companies.org/ontology#Concept"/>
  ...
</owl:members>
</owl:AllDisjointClasses>
```

Merely a declaration of a consistency requirement without standardized reporting semantics

SPARQL

```
SELECT ?entity ?t1 ?t2 {
  ?entity a ?t1 .
  ?entity a ?t2 .
  FILTER (?t1 != ?t2 && ?t1 != onto:Entity && ?t2 != onto:Entity) .
  ?t1 rdfs:subClassOf onto:Entity .
  ?t2 rdfs:subClassOf onto:Entity .
  FILTER NOT EXISTS { ?t1 a onto:Classifier } .
  FILTER NOT EXISTS { ?t2 a onto:Classifier }
}
```

An operational query for entities with more than one entity type

End of presentation

- Thank you!
- Please send feedback any time.
- Please feel free to reuse the material.
- Do you see any means of collaboration?