TECHNISCHE
UNIVERSITÄT
DRESDEN

MODELPLEX

REWERSE©
reasoning on the web

Faculty of Computer Science, Institute of Software and Multimedia Technology, Software Technology Group

Reuse for the Reuse-Agnostic

– Adding Modularity to Your Language of Choice

http://reuseware.org

Jakob Henriksson, Jendrik Johannes, Steffen Zschaler and Uwe Aßmann
Technische Universität Dresden, Chair of Software Engineering

Queens University, Feb 13, 2009

CS Department in Dresden

OUTPUT Demo Day of the Department

TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science, Institute of Software and Multimedia Technology, Software Technology Group

Bierkasten Research

– or: how to get rid of the C preprocessor

http://reuseware.org

Jakob Henriksson, Jendrik Johannes, Steffen Zschaler and Uwe Aßmann

Bierkasten - An Old One

Stefan Kühn, GNU ODL, http://upload.wikimedia.org/wikipedia/commons/0/04/Bierkasten.jpg

Bierkastens Talk about Reuse

Chumwa CCBYSA-3.0 https://de.wikipedia.org/wiki/Datei:BierkistenAufPaletten.jpg

TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science, Institute of Software and Multimedia Technology, Software Technology Group

Getting Rid of the CPP

– Adding Modularity to Your Language of Choice

http://reuseware.org

Jakob Henriksson, Jendrik Johannes, Steffen Zschaler and Uwe Aßmann

Which Kind of Language is That?

```
#ifdef _mymod_h
#define _mymod_h
#include mymod.h

#define max

#ifdef SUN
#define REGW
#else
#define REGWINDOW 0
#endif
#endif
```

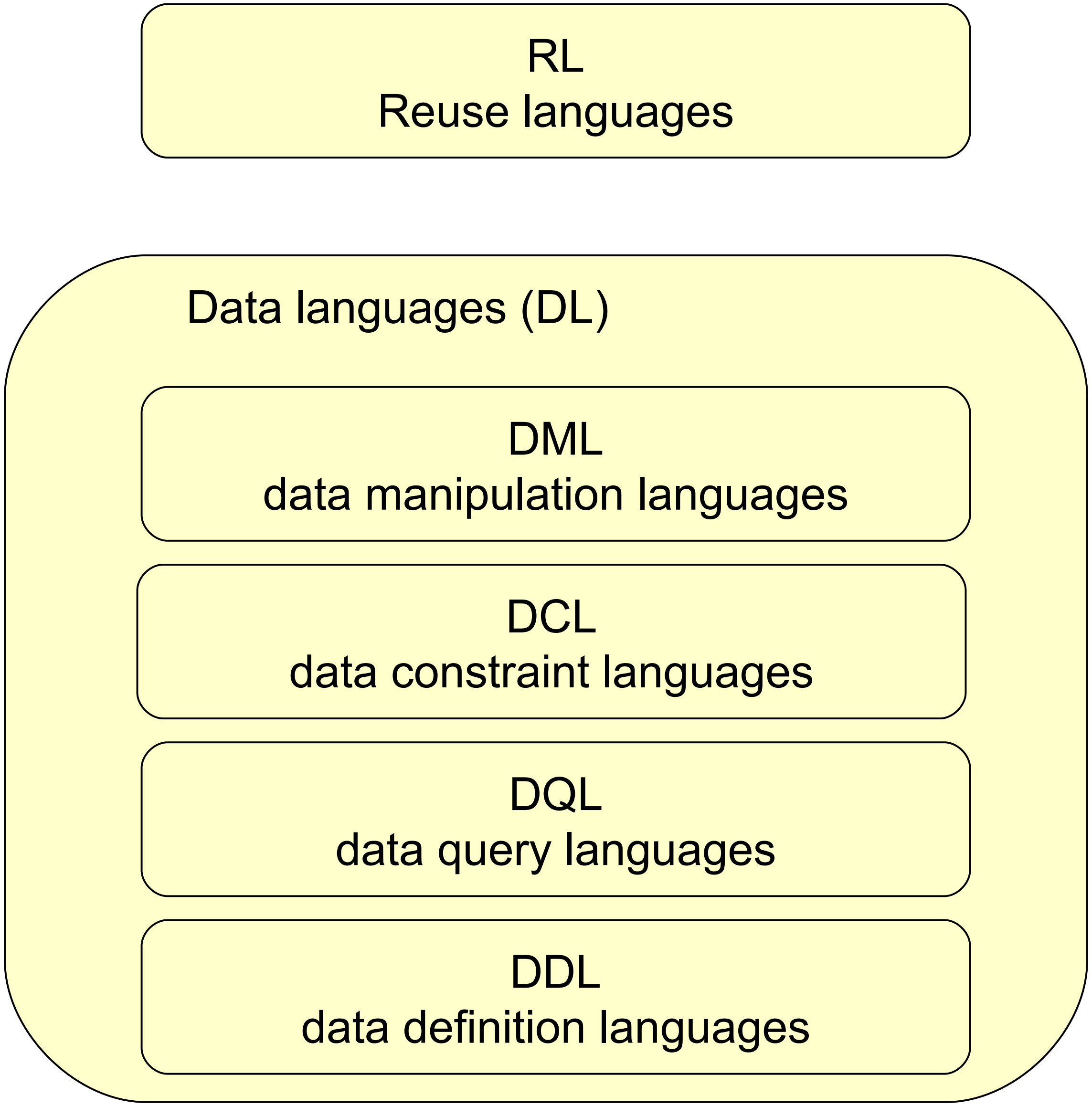
Why do we use the C preprocessor?

The Vision: Heterogenous Programs

- What is reuse code? What is algorithmic code?

```
use SQL.5.0 for query
use Modula.2.0 for scopes
use C++.2040 for class templates
use BETA for slots


template class S, DB {
  IMPLEMENTATION MODULE WebServer<S>;
    PROCEDURE <<..>> END;
  BEGIN
    S: servletGenerator = DB.init;
    R: relation = select all from DB
                       where Person == “Assmann”;
  END
}
```

A Language Uses Different Sublanguages (Language Components)



RL
Reuse languages

Data languages (DL)

DML
data manipulation languages

DCL
data constraint languages

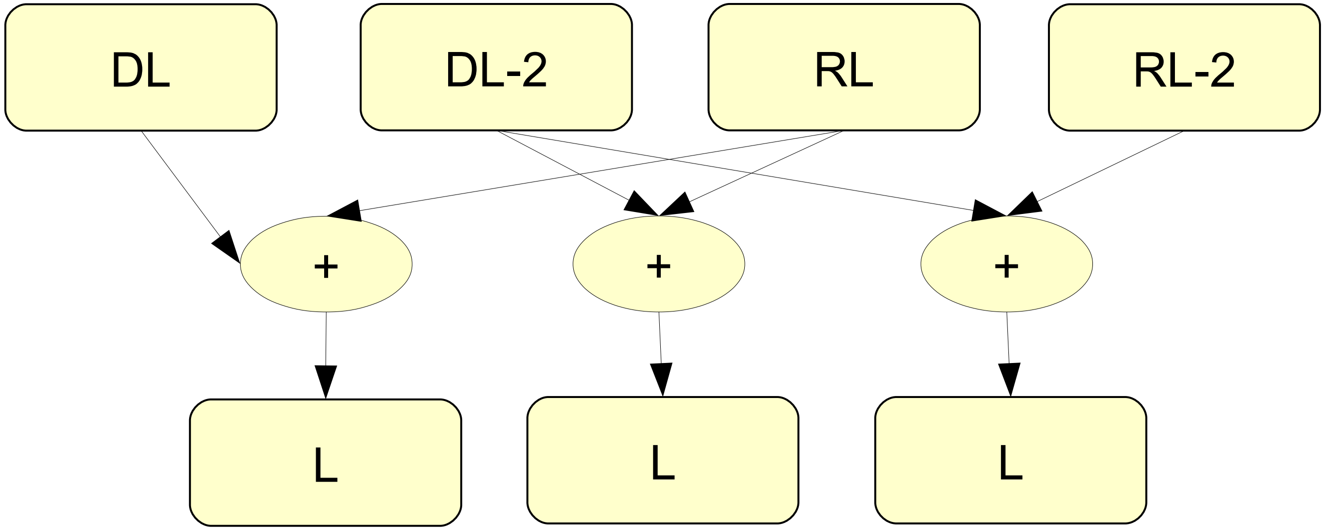
DQL
data query languages

DDL
data definition languages

import
module
< >
extends

Reuse Language

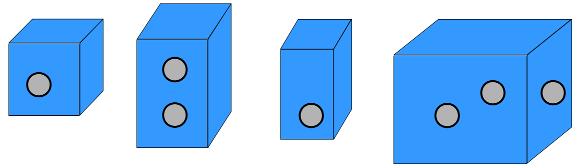
- A **(program) reuse language** is a language that describes how programs written in a DL should be reused
  - As a component, it can be composed with DL language components
  - possible in language variations



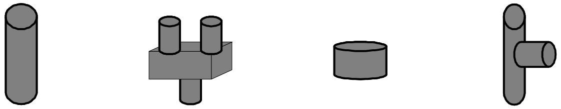
RL vs ADL

- But I thought, architectural description languages (ADL) were about reuse...

ADL Construct Black-Box Systems

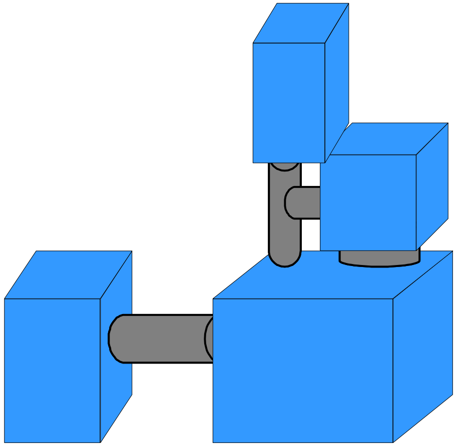


Components

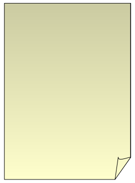


Basic composition technique (Composers)

Black-box composition



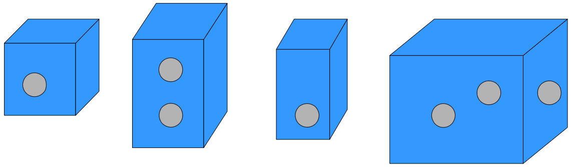
System constructed in a component- and composition-based architecture



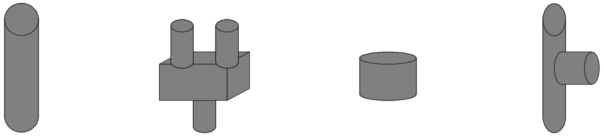
Composition recipe

Reuse is not black-box!

Invasive Software Composition (ISC) Construct Grey-Box Systems



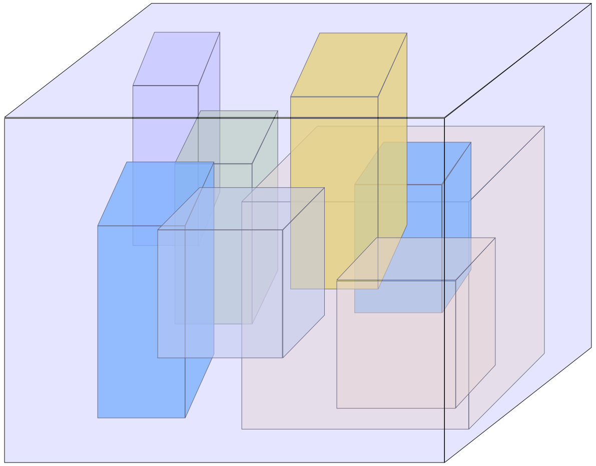
Components



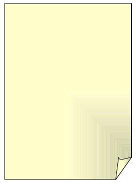
Composition Operators

Invasive Software Composition

(Grey-box Composition)



System with an Integrated Architecture



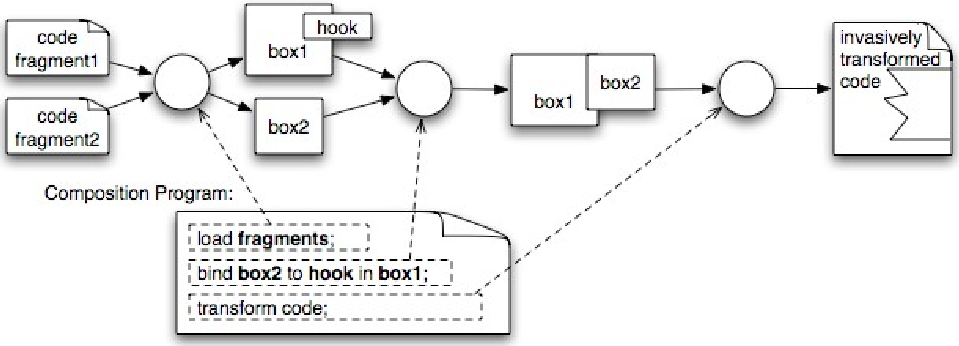
Reuse Skript

(Composition script)

ISC is not so easy to use

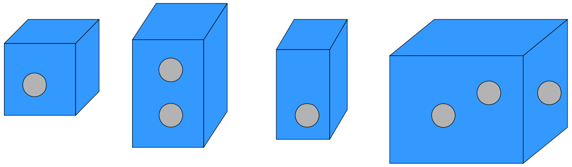
Invasive Software Composition: Summary

- Fragment-based grey-box composition technique
- Variation points
  - ***Slots*** for genericity
  - ***Hooks*** for extensibility
- **Primitive** composition operators
  - ***Bind()*** operating on slots
  - ***Extend()*** operating on hooks
- Models ADL and other programming paradigms

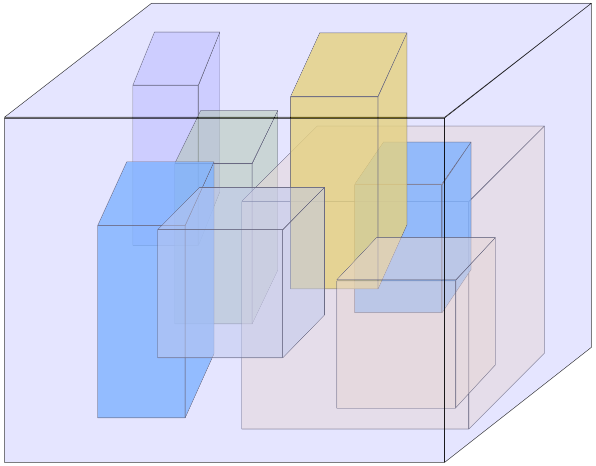


Hypothesis: RL are a special form of ISC

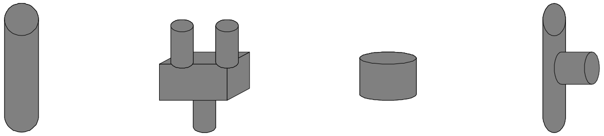
...but reuse programs are embedded, to make it simpler



Components with embedded reuse scripts (embedded composition scripts)

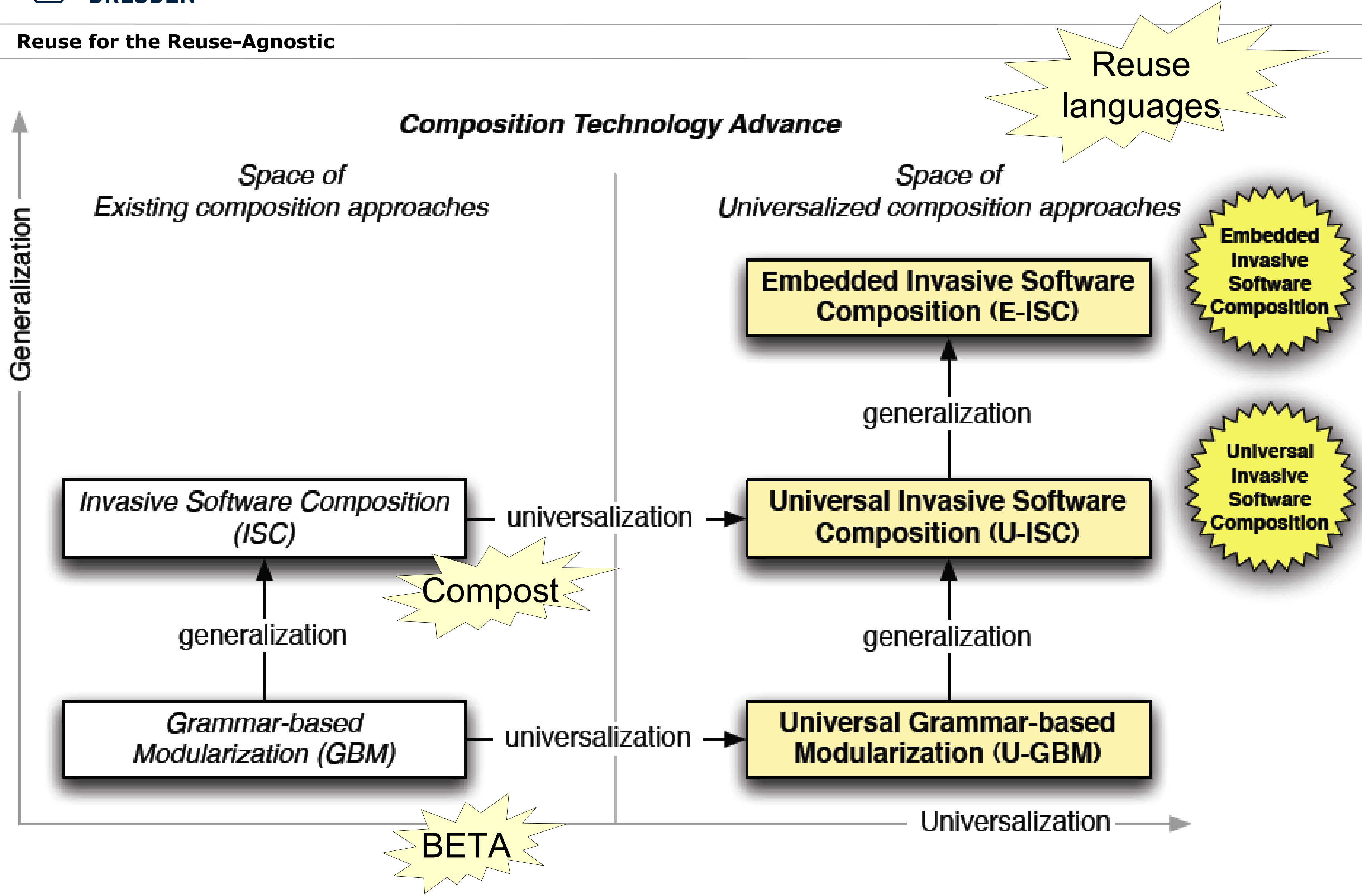


System with an Integrated Architecture



Composition Operators

Universal Invasive Composition



Reuse languages
Composition Technology Advance
Generalization
Space of
Existing composition approaches
Space of
Universalized composition approaches
Embedded Invasive Software Composition (E-ISC)
Embedded Invasive Software Composition
generalization
Invasive Software Composition (ISC)
universalization
Universal Invasive Software Composition (U-ISC)
Universal Invasive Software Composition
Compost
generalization
generalization
Grammar-based Modularization (GBM)
universalization
Universal Grammar-based Modularization (U-GBM)
BETA
Universalization

Example

Xcerpt,

a Module-Agnostic Query Language,

gets Modules

TECHNISCHE UNIVERSITÄT DRESDEN

Xcerpt: Introduction (1/2)

- Data Query and Transformation Language for XML and RDF/OWL
  [Schaffert*et al*., 2004]
- Data terms
  - Represent XML documents

    ```
    <book><title>T</title><author>A</author></book>
     book [ title [ “T” ], author [ “A” ] ]
    ```
- Query terms
  - Patterns matching data terms resulting in answer substitutions

  book [title [var X], author [var Y]] → { X / “T”, Y / “A” }
- Construct terms
  - Data terms with variables to be instantiated
  - Builds data terms by applying answer substitutions

Xcerpt: Introduction (2/2)

Xcerpt programs:

- A set of rules of the form:

```
CONSTRUCT                     GOAL
    head                          head
FROM                          FROM
    body                          body
END                           END
```

- **head** is a construct term
- **body** is a set of query terms connected using connectives:
  - **AND** or **OR**

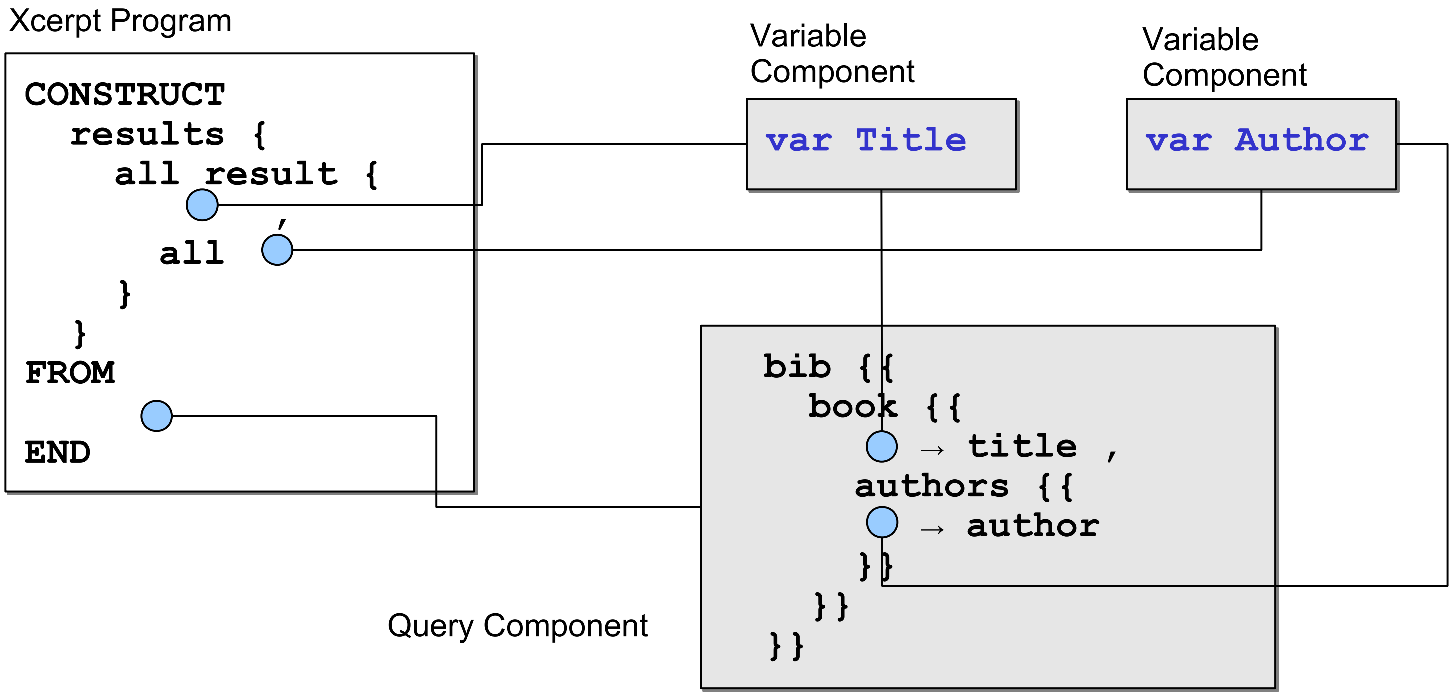
A Language without Reuse: Xcerpt

```
CONSTRUCT
  results {
    all result {
      var Title,
      all var Author
    }
  }
FROM
    bib {{
      book {{
        var Title → title ,
        authors {{
           var Author → author
        }}
      }}
    }}
END
```

```
CONSTRUCT
  results {
    all result {
      all var Title,
      var Author
    }
  }
FROM
    bib {{
      book {{
        var Title → title ,
        authors {{
           var Author → author
        }}
      }}
    }}
END
```

Possible Reuse in Xcerpt

- Overlapping fragments could be factored out



Xcerpt Program

```
CONSTRUCT
  results {
    all result {
      ●
             ,
      all  ●
    }
  }
FROM
     ●
END
```

Variable Component

```
var Title
```

Variable Component

```
var Author
```

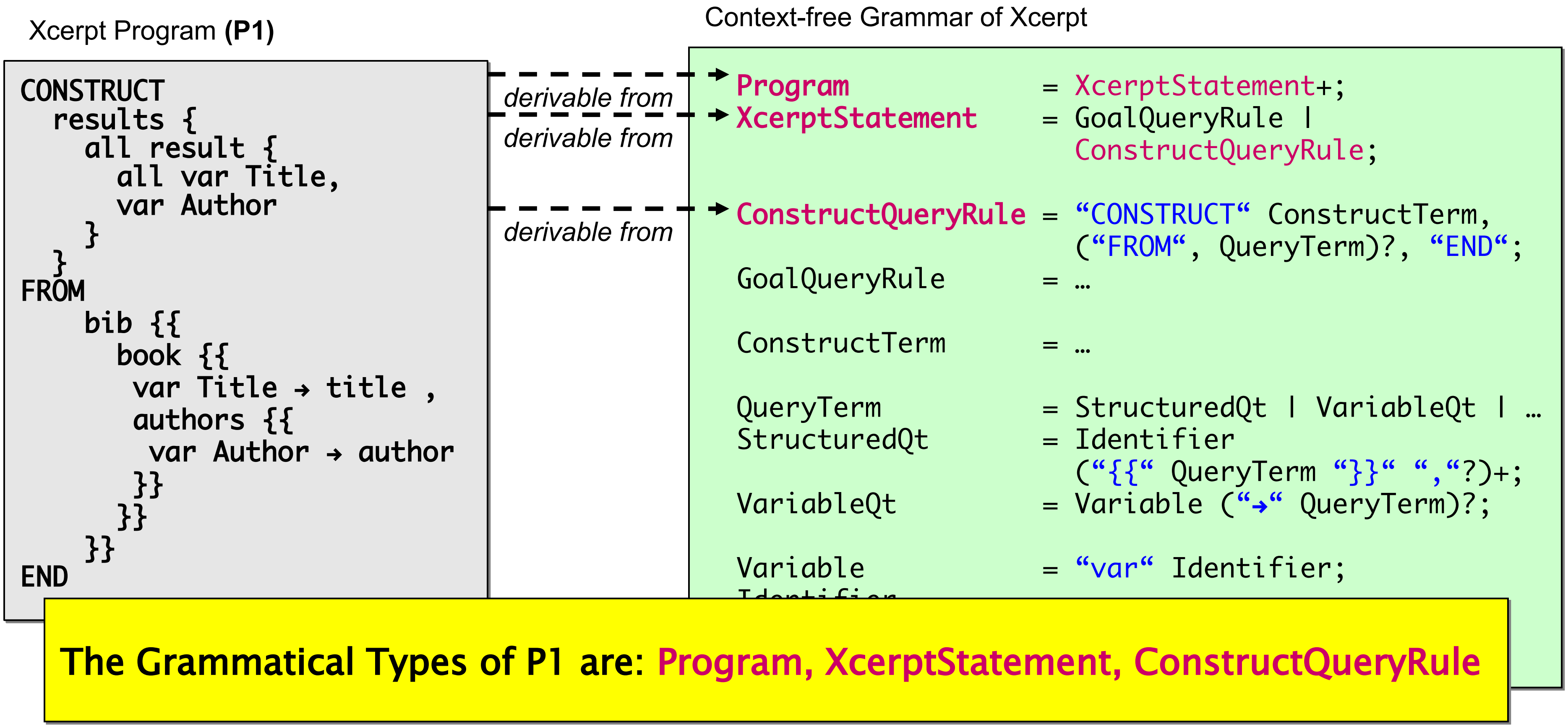
Query Component

```
bib {{
  book {{
     ● → title ,
    authors {{
     ● → author
    }}
  }}
}}
```

● = variation point

Grammatical Types for Grammar-Based Modularization

- BETA style: separate compilation of all parts of a program



Xcerpt Program **(P1)**

```
CONSTRUCT
  results {
    all result {
      all var Title,
      var Author
    }
  }
FROM
    bib {{
      book {{
       var Title → title ,
       authors {{
        var Author → author
       }}
      }}
    }}
END
```

Context-free Grammar of Xcerpt

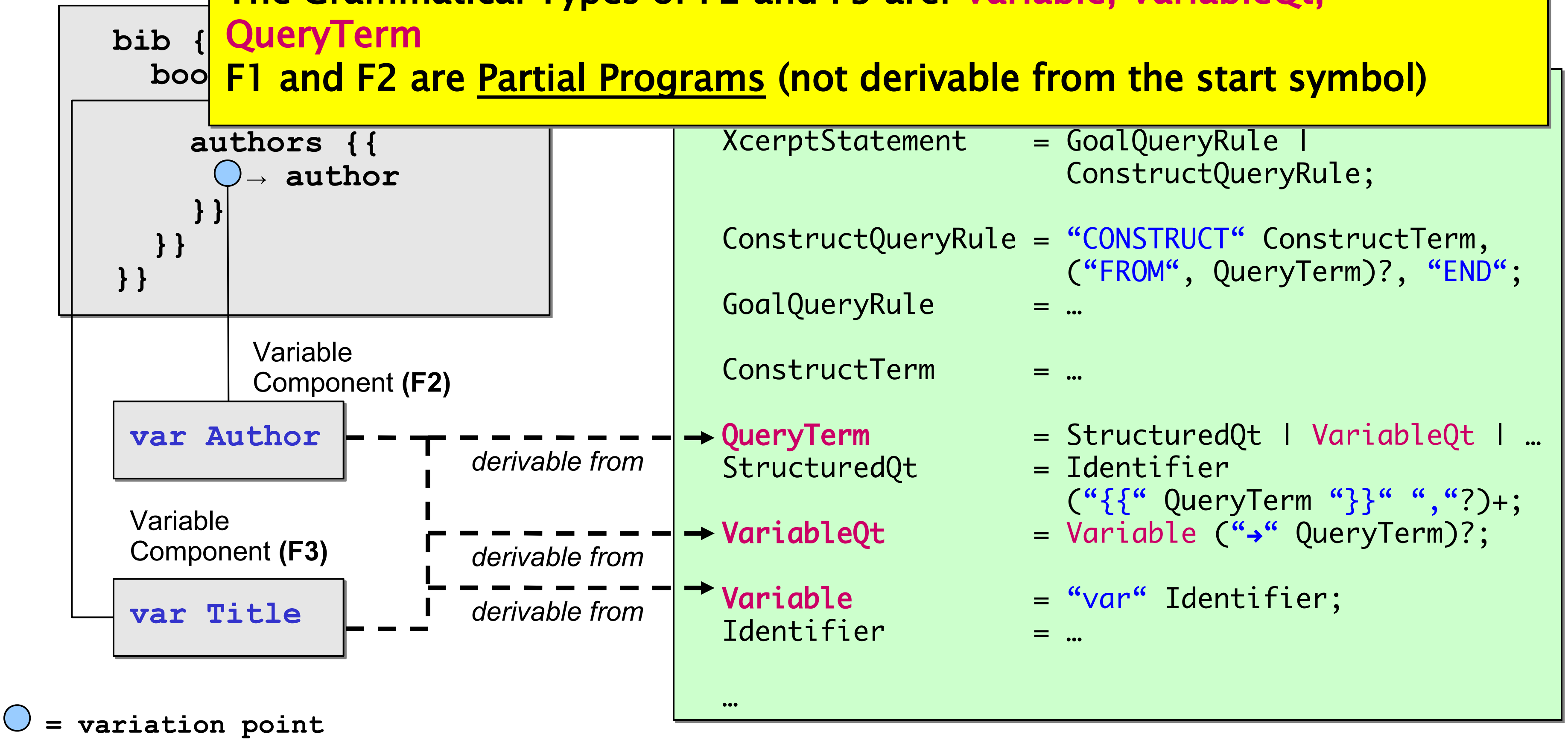
```
Program             = XcerptStatement+;
XcerptStatement     = GoalQueryRule |
                      ConstructQueryRule;

ConstructQueryRule  = “CONSTRUCT“ ConstructTerm,
                      (“FROM“, QueryTerm)?, “END“;
GoalQueryRule       = …

ConstructTerm       = …

QueryTerm           = StructuredQt | VariableQt | …
StructuredQt        = Identifier
                      (“{{“ QueryTerm “}}“ “,“?)+;
VariableQt          = Variable (“→“ QueryTerm)?;

Variable            = “var“ Identifier;
Identifier
```

The Grammatical Types of P1 are: Program, XcerptStatement, ConstructQueryRule

Partial Programs (Fragments)

The Grammatical Types of F2 and F3 are: Variable, VariableQt, QueryTerm

F1 and F2 are <u>Partial Programs</u> (not derivable from the start symbol)



```
bib {
  boo
      authors {{
        ● → author
      }}
    }}
}}
```

Variable Component **(F2)**

```
var Author
```

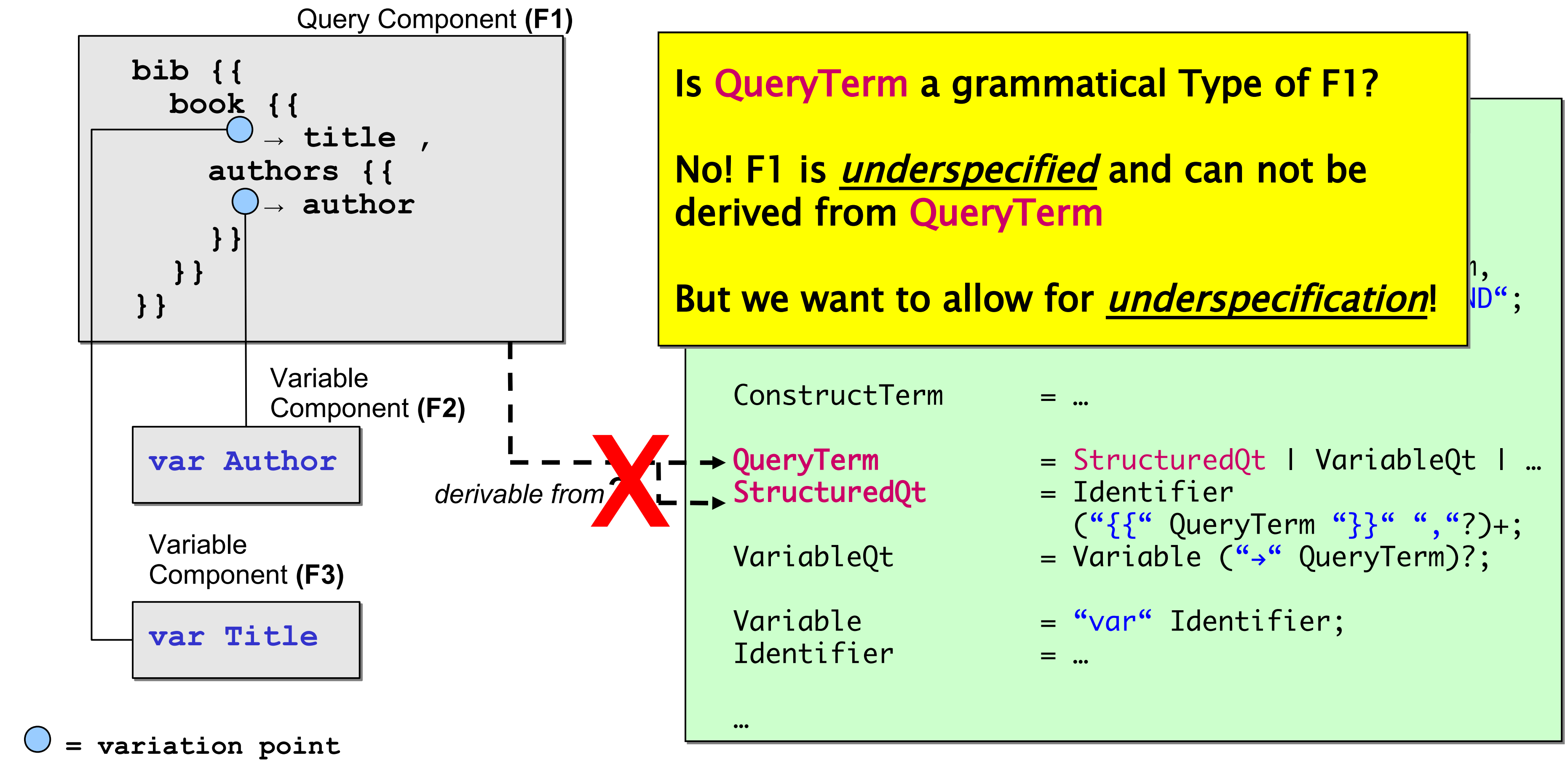
Variable Component **(F3)**

```
var Title
```

```
XcerptStatement     = GoalQueryRule |
                      ConstructQueryRule;

ConstructQueryRule = “CONSTRUCT“ ConstructTerm,
                      (“FROM“, QueryTerm)?, “END“;
GoalQueryRule       = …

ConstructTerm       = …

QueryTerm           = StructuredQt | VariableQt | …
StructuredQt        = Identifier
                      (“{{“ QueryTerm “}}“ “,“?)+;
VariableQt          = Variable (“→“ QueryTerm)?;

Variable            = “var“ Identifier;
Identifier          = …

…
```

● = **variation point**

Underspecified Partial Programs (Generic Fragments)



Query Component **(F1)**

```
bib {{
  book {{
   → title ,
    authors {{
      → author
    }}
  }}
}}
```

Variable Component **(F2)**

```
var Author
```

Variable Component **(F3)**

```
var Title
```

Is QueryTerm a grammatical Type of F1?

No! F1 is ***underspecified*** and can not be derived from QueryTerm

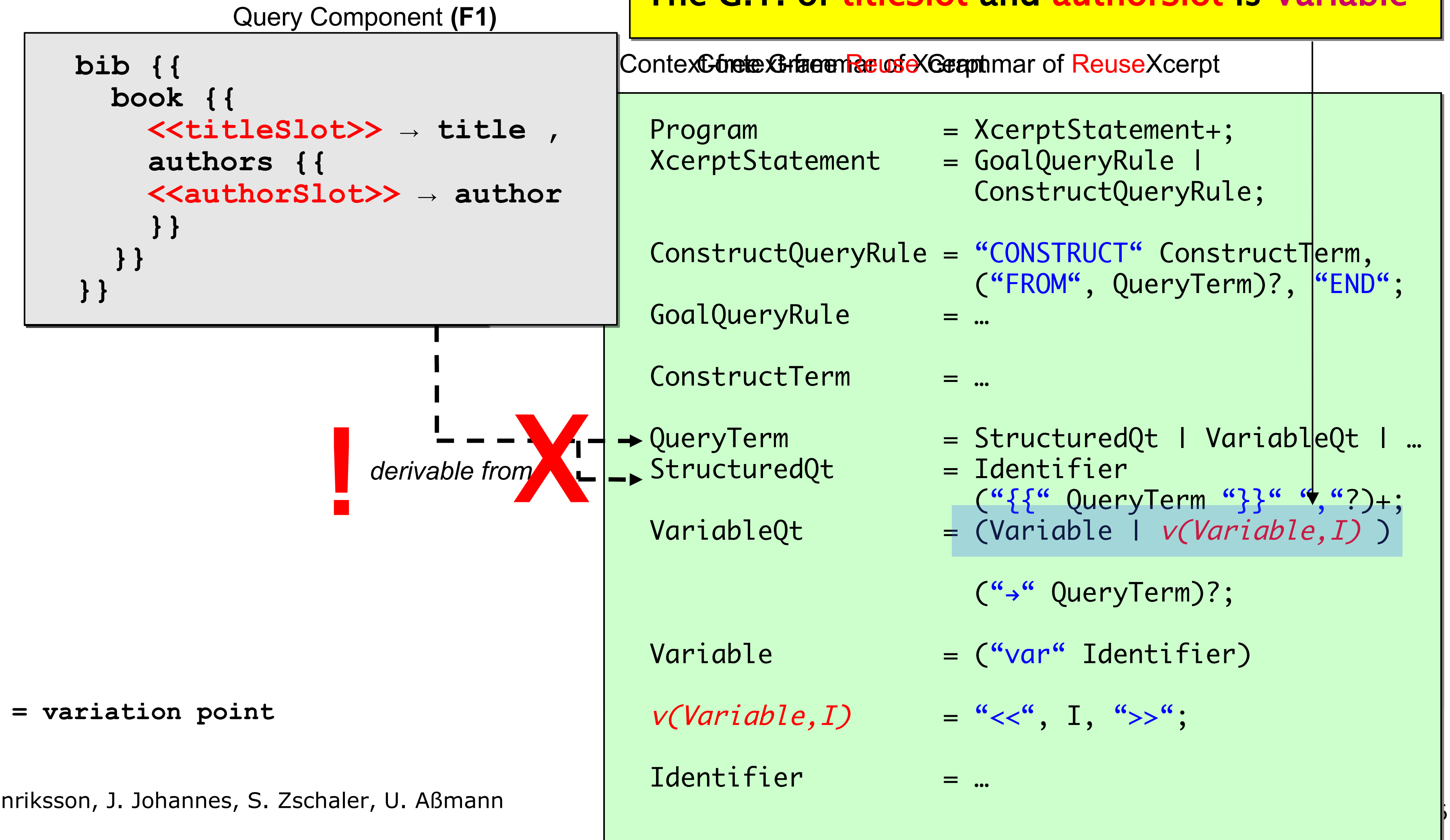
But we want to allow for ***underspecification***!

```
                                                  n,
                                                  ND“;

ConstructTerm      = …

QueryTerm          = StructuredQt | VariableQt | …
StructuredQt       = Identifier
                     (“{{“ QueryTerm “}}“ “,“?)+;
VariableQt         = Variable (“→“ QueryTerm)?;

Variable           = “var“ Identifier;
Identifier         = …

…
```

Variation Points in Generic Fragments

Variation Points have a Grammatical Type

The G.T. of titleSlot and authorSlot is Variable



Query Component **(F1)**

```
bib {{
  book {{
    <<titleSlot>> → title ,
    authors {{
    <<authorSlot>> → author
    }}
  }}
}}
```

Context-Free Grammar of ReuseXcerpt

```
Program              = XcerptStatement+;
XcerptStatement      = GoalQueryRule |
                       ConstructQueryRule;

ConstructQueryRule = "CONSTRUCT" ConstructTerm,
                     ("FROM", QueryTerm)?, "END";
GoalQueryRule        = …

ConstructTerm        = …

QueryTerm            = StructuredQt | VariableQt | …
StructuredQt         = Identifier
                       ("{{" QueryTerm "}}" ","?)+;
VariableQt           = (Variable | v(Variable,I) )

                       ("→" QueryTerm)?;

Variable             = ("var" Identifier)

v(Variable,I)        = "<<", I, ">>";

Identifier           = …
```

● = variation point

Beyond BETA: Reuse Grammars

- Reuse Grammars specify Reuse Languages
- Th

A Reuse Grammar G_I is the result of a transformation of a Core Grammar G: This transformation is…

…*preservative:* Any String derivable from a non-terminal in G can still be derived from the same non-terminal in G_I

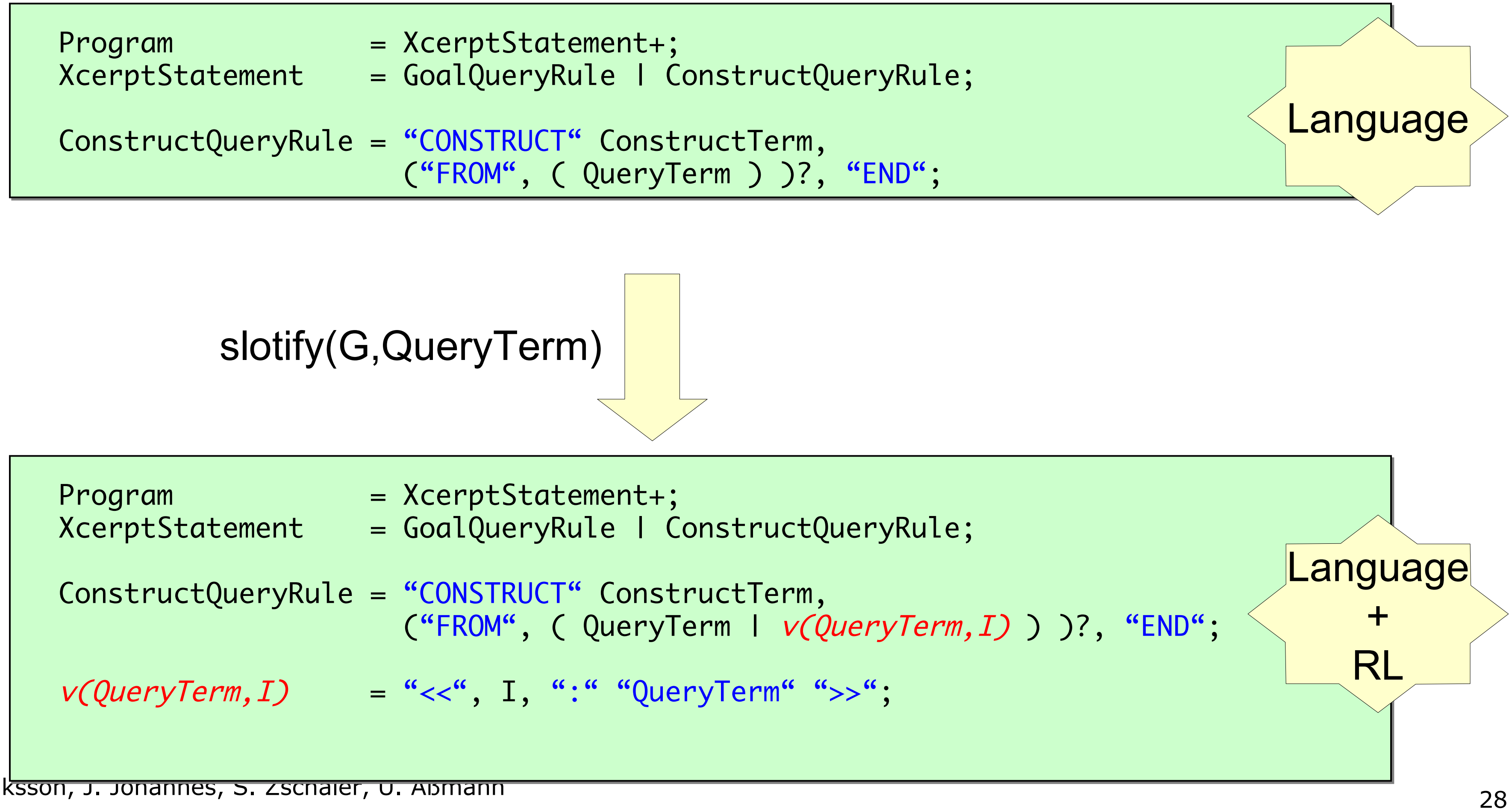
…*type preservative:* In production rules, variation points are only introduced as alternatives to their types

Underspecified and/or partial programs wrt. G that are valid programs wrt. G_I are valid *Fragments* wrt. G

```
Prog
Xcer

Cons

Goal

Cons

Quer
Stru

Vari


Variable             = (“var“ Identifier)

v(Variable,I)        = “<<“, I, “:“ “Variable“  “>>“;
v(QueryTerm,I)       = “<<“, I, “:“ “QueryTerm“ “>>“;

Identifier           = …

…
```

Slotify – a Grammar Transformer

- Slotify is a grammar transformer, designating nonterminals for the creation of slots in reuse grammars
- Slotify adds a reuse language to a language

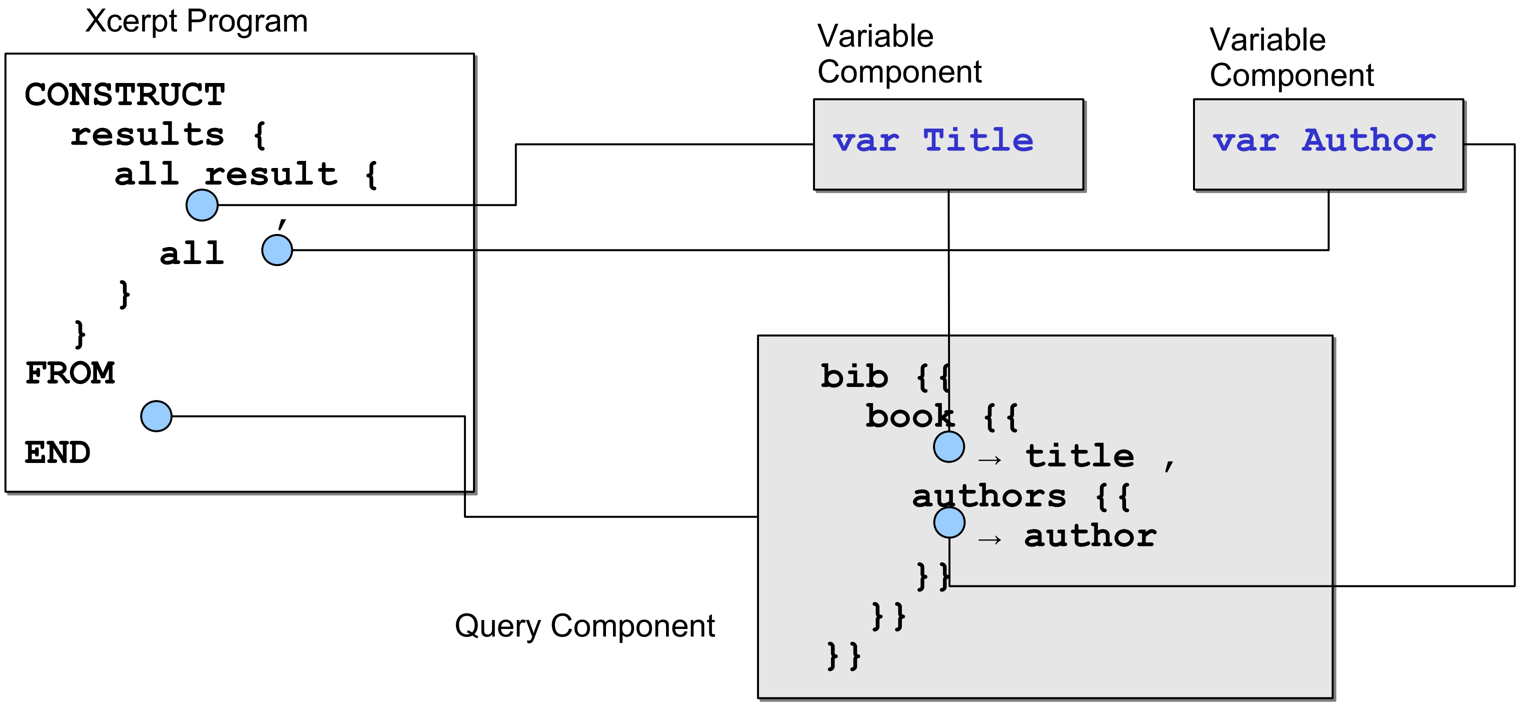


```
Program           = XcerptStatement+;
XcerptStatement   = GoalQueryRule | ConstructQueryRule;

ConstructQueryRule = “CONSTRUCT“ ConstructTerm,
                    (“FROM“, ( QueryTerm ) )?, “END“;
```

```
Program           = XcerptStatement+;
XcerptStatement   = GoalQueryRule | ConstructQueryRule;

ConstructQueryRule = “CONSTRUCT“ ConstructTerm,
                    (“FROM“, ( QueryTerm | v(QueryTerm,I) ) )?, “END“;

v(QueryTerm,I)     = “<<“, I, “:“ “QueryTerm“ “>>“;
```

Composition Operators & Type Safety

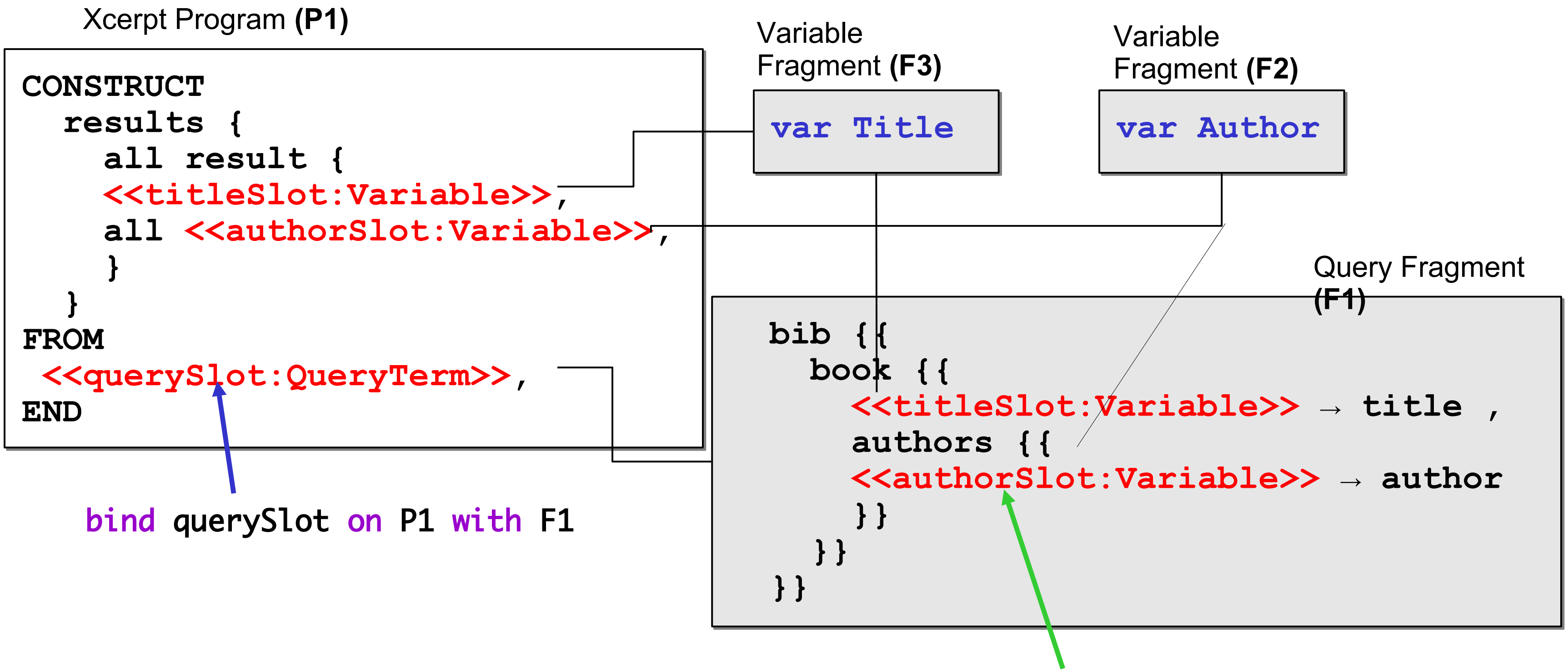
- Slots are bound *type-safe*



J. Henriksson, J. Johannes, S. Zschaler, U. Aßmann

Composition Operators & Type Safety

Primitive Composition Operators:
– Take two fragments (F1 & F2) and an variation point (authorSlot) in F1 as argument
– replace authorSlot with F2 (type-safe)
– replace authorSlot in F1 with F2 (type-safe)



Xcerpt Program **(P1)**

```
CONSTRUCT
  results {
    all result {
    <<titleSlot:Variable>>,
    all <<authorSlot:Variable>>,
    }
  }
FROM
 <<querySlot:QueryTerm>>,
END
```

Variable Fragment **(F3)**

```
var Title
```

Variable Fragment **(F2)**

```
var Author
```

Query Fragment **(F1)**

```
bib {{
  book {{
    <<titleSlot:Variable>> → title ,
    authors {{
    <<authorSlot:Variable>> → author
    }}
  }}
}}
```

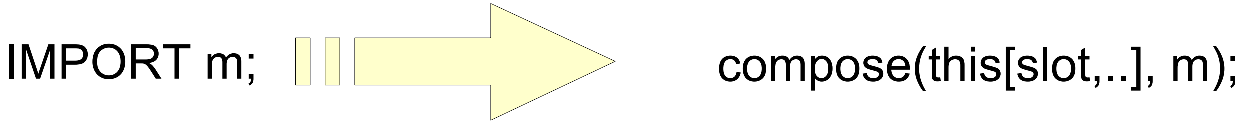
bind querySlot on P1 with F1

bind authorSlot on F1 with F2

Active Syntax for Embedded Reuse Languages

- How to embed compositions (reuse statements) into a core language?

- Answer: by active syntax
  - Keywords trigger compositions
  - Parser starts compositions



IMPORT m;

compose(this[slot,..], m);

where compose is
a composition operator

- Reusing a module for transitive closure for OWL transitive closure

Configurable Xcerpt Module:
/subClassOf.mxcerpt

```
CONSTRUCT  // Transitive closure query
    inferredSubClassOf [
      all subClassOf [ var Subclass, var Superclass ]
    ]
FROM
  or {
    declsubclassof [ var Subclass, var Superclass ],
    and {
      declsubclassof [ var Subclass
      declsubclassof [ var Z, var S
    }
  }
END

CONSTRUCT  // Constructing base rel
  declsubclassof [ var Subclass, va
FROM
  <<rootNode>> [[
    Class {
      id { var Subclass },
      subClassOf {
        about { var Superclass }
      }
    }
  ]]
END
```

Xcerpt Program
with IMPORT

```
IMPORT
   /subClassOf.mxcerpt [ bind(rootNode, 'owl') ]
END

GOAL
   result [ all var X ]
FROM
   var X -> inferredSubClassOf [[ ]]
END

CONSTRUCT // Using transitive closure on OWL classes
   owl [
     Class [ id [ "SportsEquipment" ] ],
     Class [ id [ "TennisRacket" ],
             subClassOf [ about [ "SportsEquipment" ] ] ],
     Class [ id [ "WilsonTennisRacket" ],
             subClassOf [ about [ "TennisRacket" ] ] ]
   ]
END
```

Active Syntax Expands to Composition Scripts

Configurable Xcerpt Module:
/subClassOf.mxcerpt

```
CONSTRUCT
    inferredSubClassOf [
      all subClassOf [ var Subclass, var Superclass ]
    ]
FROM
  or {
    declsubclassof [ var Subclass, var Superclass ],
    and {
      declsubclassof [ var Subclass
      declsubclassof [ var Z, var S
    }
  }
END

CONSTRUCT
  declsubclassof [ var Subclass, va
FROM
  <<rootNode>> [[
    Class {
      id { var Subclass },
      subClassOf {
        about { var Superclass }
      }
    }
  ]]
END
```

Xcerpt Program
with IMPORT

```
COMPOSITION SCRIPT BEGIN
   include(subClassOf.mxcerpt [ bind(rootNode, 'owl') ];
END

GOAL
   result [ all var X ]
FROM
   var X -> inferredSubClassOf [[ ]]
END

CONSTRUCT
   owl [
     Class [ id [ "SportsEquipment" ] ],
     Class [ id [ "TennisRacket" ],
             subClassOf [ about [ "SportsEquipment" ] ] ],
     Class [ id [ "WilsonTennisRacket" ],
             subClassOf [ about [ "TennisRacket" ] ] ]
   ]
END
```

Composition Scripts Compose Modules as Fragments

```
CONSTRUCT
    inferredSubClassOf [
      all subClassOf [ var Subclass, var Superclass ]
    ]
FROM
  or {
    declsubclassof [ var Subclass, var Superclass ],
    and {
      declsubclassof [ var Subclass, var Z ],
      declsubclassof [ var Z, var Superclass ]
    }
  }
END

CONSTRUCT
  declsubclassof [ var Subclass, va
FROM
  owl [[
    Class {
      id { var Subclass },
      subClassOf {
        about { var Superclass }
      }
    }
  ]]
END
```

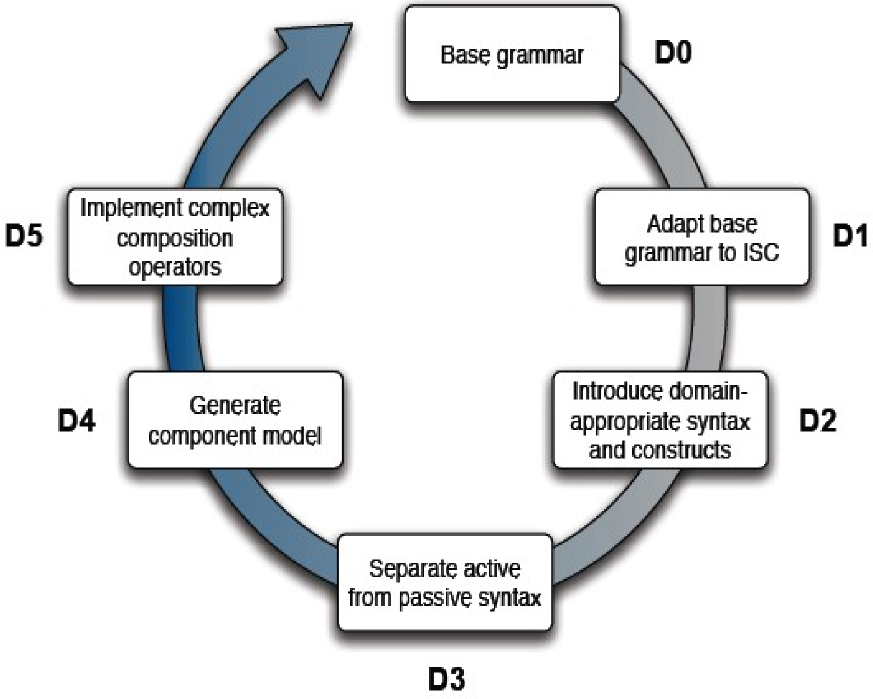
Just a preprocessor?

NO -
type-safe !

```
GOAL
   result [ all var X ]
FROM
   var X -> inferredSubClassOf [[ ]]
END

CONSTRUCT
   owl [
     Class [ id [ "SportsEquipment" ] ],
     Class [ id [ "TennisRacket" ],
             subClassOf [ about [ "SportsEquipment" ] ] ],
     Class [ id [ "WilsonTennisRacket" ],
             subClassOf [ about [ "TennisRacket" ] ] ]
   ]
END
```

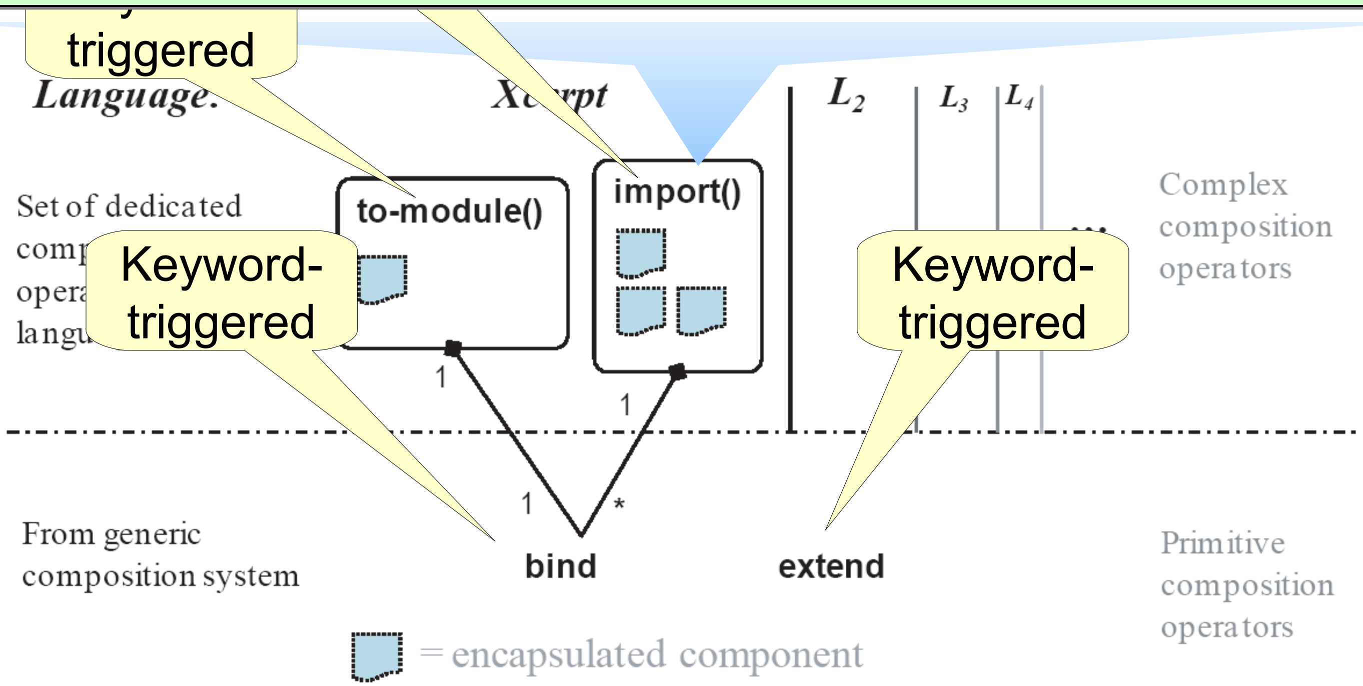
How to do this universally?

- Given a grammar of a language
- Construct a reuse grammar for the reuse language
- Generate a composition system from it
- Define active syntax for it
- ==> a type-safe reuse-language preprocessor


Base grammar
D0
Adapt base grammar to ISC
D1
Introduce domain-appropriate syntax and constructs
D2
Separate active from passive syntax
D3
Generate component model
D4
Implement complex composition operators
D5

Complex Composition Operators Triggered by

```
define composer modularxcerpt.ImportComposer(moduleLocation, args) {

    fragmentlist xcerpt.XcerptProgram module = ->moduleLocation;

    foreach(element : args) {
        bind ->element.slot on module with element.value;
    }

    return module.statements;
}
```


triggered
Language:
Xcerpt
L_2
L_3
L_4
Set of dedicated composition operators language
to-module()
import()
Keyword-triggered
Keyword-triggered
...
Complex composition operators
1
1
1
*
From generic composition system
bind
extend
Primitive composition operators
= encapsulated component

A Language for Writing Composition

```
define composer modularxcerpt.ImportComposer(moduleName) {

    fragmentlist componentmodel.Location uri    = ->moduleName;
    fragmentlist xcerpt.XcerptProgram    module = ->uri;

    if (module.statements[first] instanceof modularxcerpt.ModuleDefinition) {

      foreach (r : module.statements[first].xcerptProgram.statements) {

       if (r instanceof xcerpt.ConstructQueryRule) {

           fragmentlist xcerpt.ConstructTerm ct = r.construct;

           if (ct instanceof modularxcerpt.VisibilityConstructTerm) {
             // specific visibility
             fragmentlist modularxcerpt.Visibility visibility = ct.visibility;
           }
           else {
             // default visibility of the module
             fragmentlist modularxcerpt.Visibility visibility =
               module.statements[first].defaultVisibility;
           }

           if (visibility instanceof modularxcerpt.PublicVisibility) {
             // visibility public
             fragmentlist xcerpt.ConstructTerm ctWrapper =
               'store [ modul ["' + uri +  '"], visibility ["public"],  <<cTerm>> ]'.mxcerpt;
           }
           else {
             // visibility private
             fragmentlist xcerpt.ConstructTerm ctWrapper =
               'store [ modul ["' + uri +  '"], visibility ["private"], <<cTerm>> ]'.mxcerpt;
           }
```

Benefits of Universal Reuse Languages

- Common reuse tools for reuse-agnostic core languages
  - Forget about reuse constructs in your language, use slotification and embedded active syntax
  - hide the reuse constructs behind keywords
- Domain application engineers, language design and development becomes much simpler
  - DSL with reuse language can grow out of a core DSL, adding reuse constructs
- Embedded ISC behaves like a type-safe, language-specific preprocessor
  - normalizing the reuse language extension to the core language
  - CPP is untyped and language-agnostic
- Embedded ISC can be used to replace unsafe reuse languages
  - tailor language-specific ones

An Implementation:
The Reuseware Composition Framework

- Implements slotification and the production of reuse grammars
- Framework and GUI to extend languages for reuse
- GUI integration into Eclipse
- Grammar-based language descriptions
  - Oriented at standard EBNF
  - Separation between abstract and concrete syntax
  - Wizards for language extension
- Composition environment generation
  - Generation of a complete composition environment for an extended language from the grammars only

  + Composition engine utilizable as pre-processor

  + Parser for extended languages (utilizing ANTLR3)

  + Eclipse-IDE including Editors with
  syntax-checking and -highlighting for extended languages

Applications

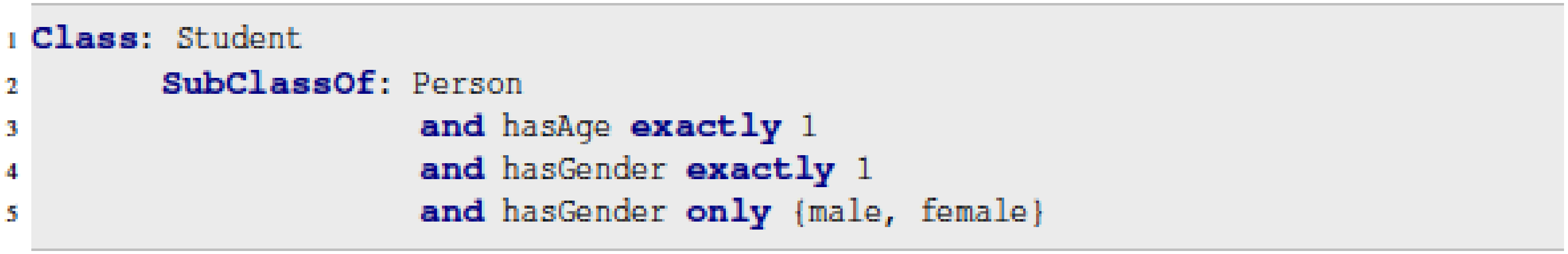
- Reuse for plain, module-less languages
  - Xcerpt
  - Prolog, Datalog
- Role models for non-role languages
  - Role models for OWL
- … many more to come...

Example 2: Role Models for Ontologies in OWL

- OWL is an ontology language based on set expressions

$$Student \sqsubseteq Person \sqcap (= 1 hasAge) \sqcap (= 1 hasGender) \sqcap \forall hasGender.\{male, female\}$$

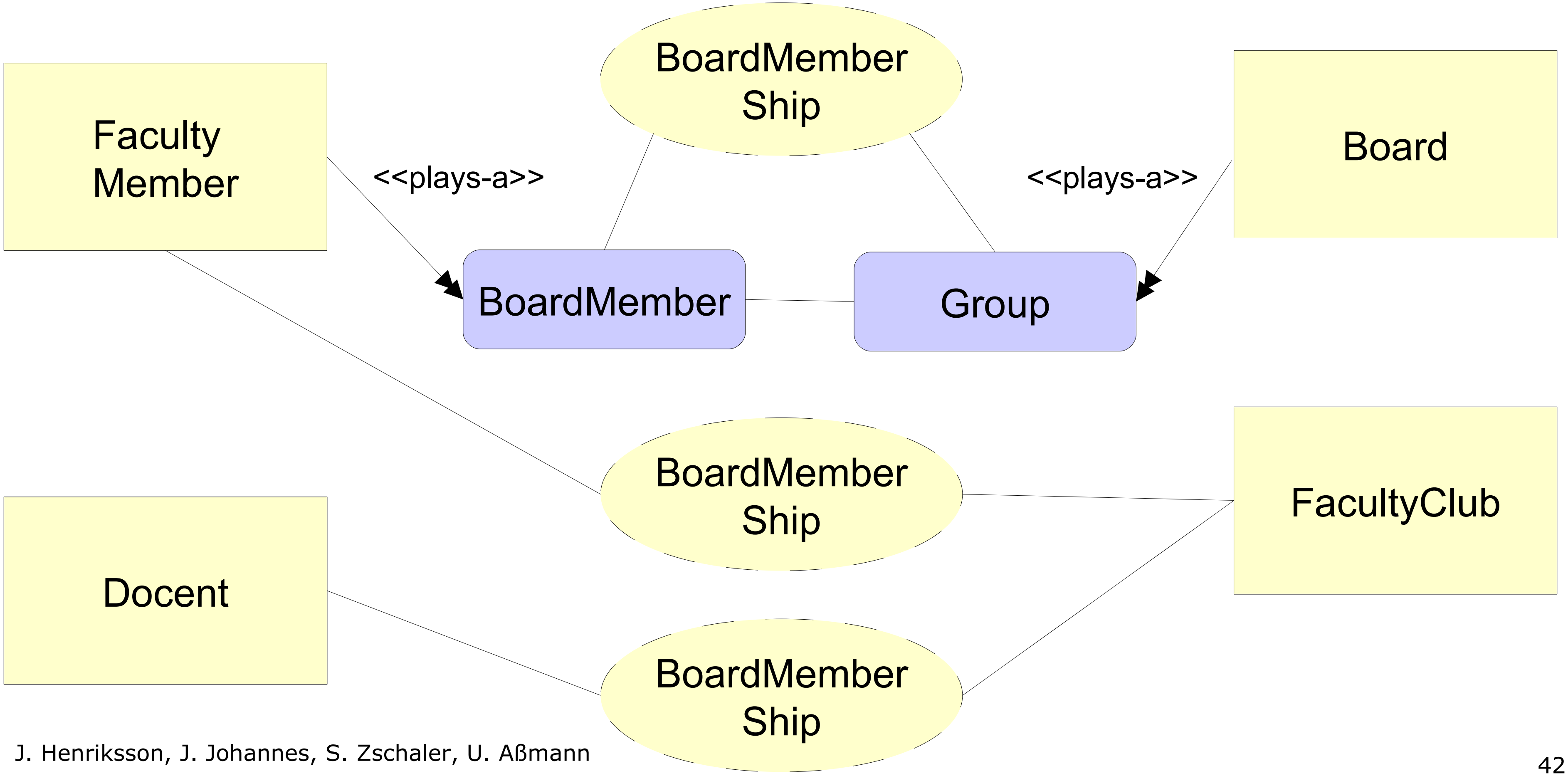
Manchester syntax for OWL



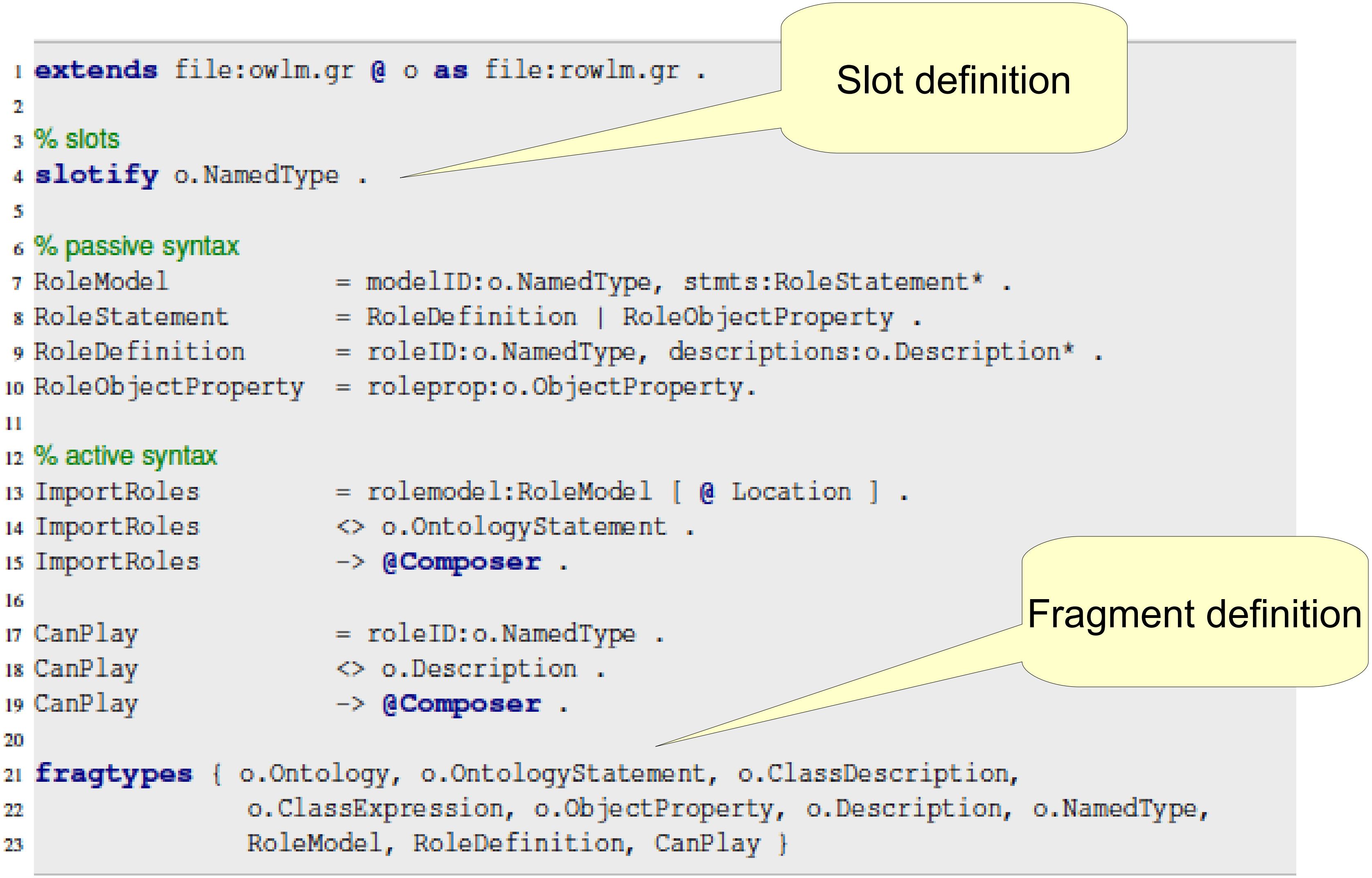
```
Class: Student
      SubClassOf: Person
                  and hasAge exactly 1
                  and hasGender exactly 1
                  and hasGender only {male, female}
```

Role Models

- Role Models are a reuse concept to isolate collaborations of classes
- They can be reused over many classes



Slotification of the OWL Grammar for an OWL Reuse Grammar



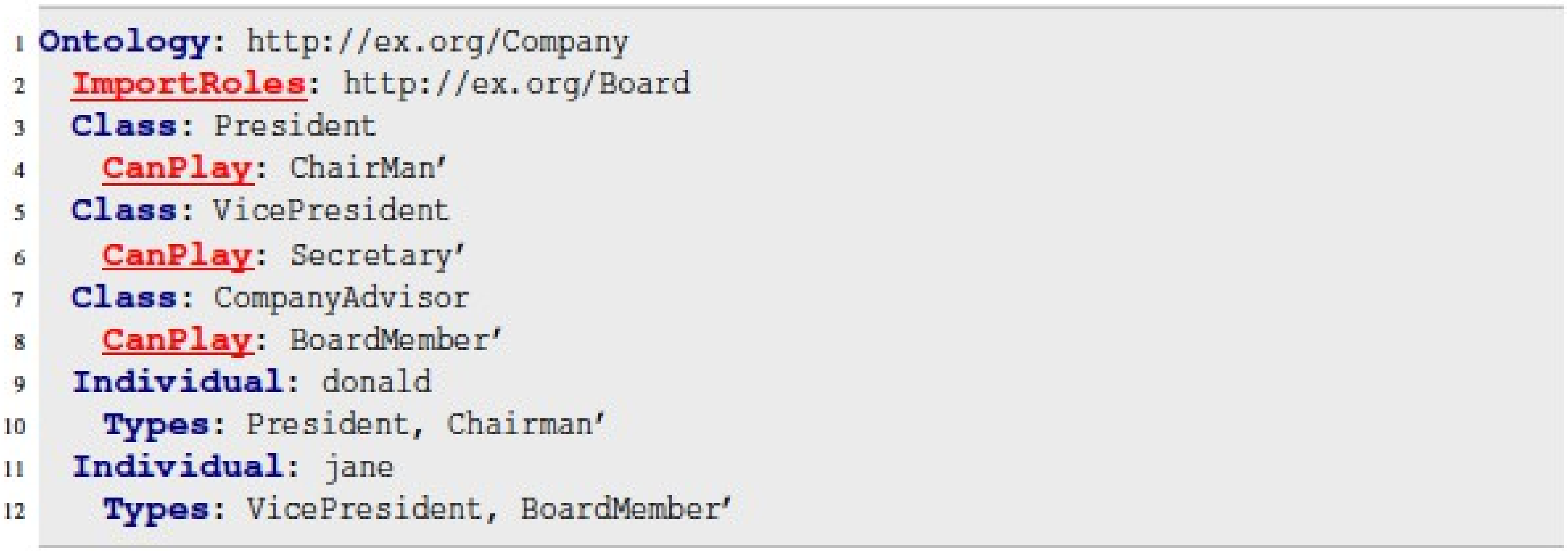
```
extends file:owlm.gr @ o as file:rowlm.gr .

% slots
slotify o.NamedType .

% passive syntax
RoleModel             = modelID:o.NamedType, stmts:RoleStatement* .
RoleStatement         = RoleDefinition | RoleObjectProperty .
RoleDefinition        = roleID:o.NamedType, descriptions:o.Description* .
RoleObjectProperty    = roleprop:o.ObjectProperty.

% active syntax
ImportRoles           = rolemodel:RoleModel [ @ Location ] .
ImportRoles           <> o.OntologyStatement .
ImportRoles           -> @Composer .

CanPlay               = roleID:o.NamedType .
CanPlay               <> o.Description .
CanPlay               -> @Composer .

fragtypes { o.Ontology, o.OntologyStatement, o.ClassDescription,
              o.ClassExpression, o.ObjectProperty, o.Description, o.NamedType,
              RoleModel, RoleDefinition, CanPlay }
```

Slot definition

Fragment definition

Active Syntax with new Keywords

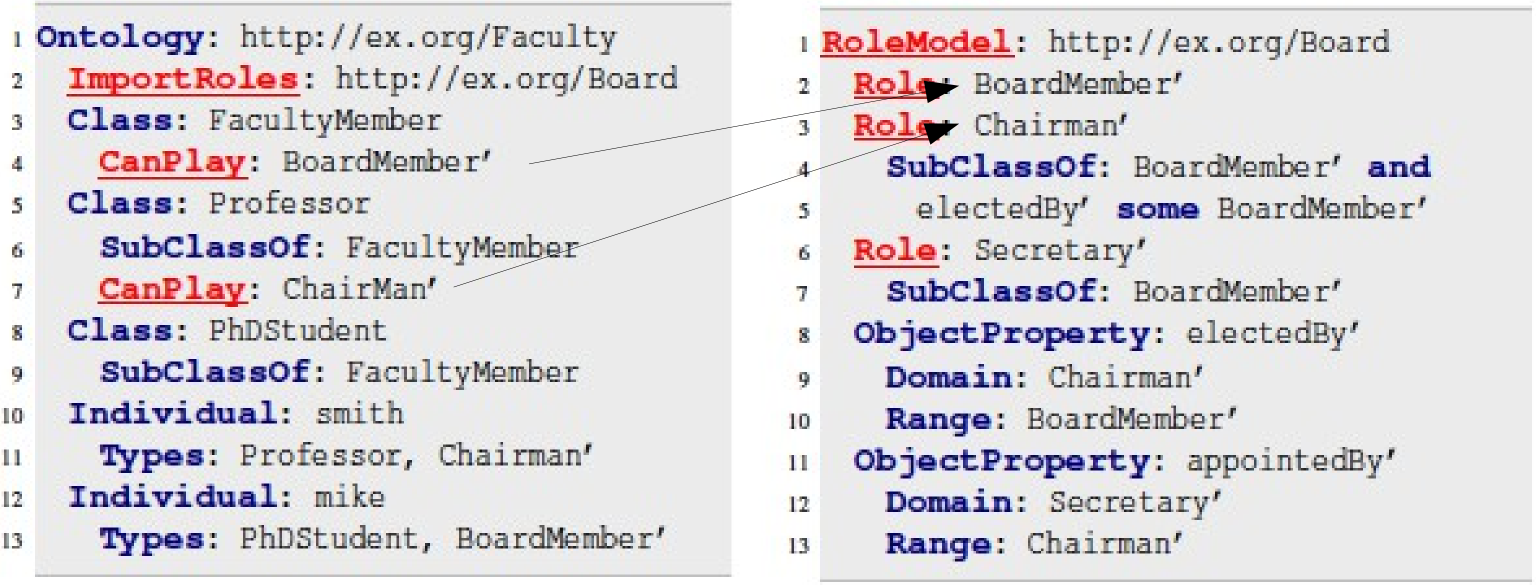
- Triggering role compositions under the hood
  - ImportRoles imports role models
  - CanPlay binds roles to classes



```
Ontology: http://ex.org/Company
  ImportRoles: http://ex.org/Board
  Class: President
    CanPlay: ChairMan'
  Class: VicePresident
    CanPlay: Secretary'
  Class: CompanyAdvisor
    CanPlay: BoardMember'
  Individual: donald
    Types: President, Chairman'
  Individual: jane
    Types: VicePresident, BoardMember'
```

OWL Role Models (Reuse-OWL)

- In Reuse-OWL, core modules can refer to role models
  - and use roles, to be played by natural types
- Role models can be reused

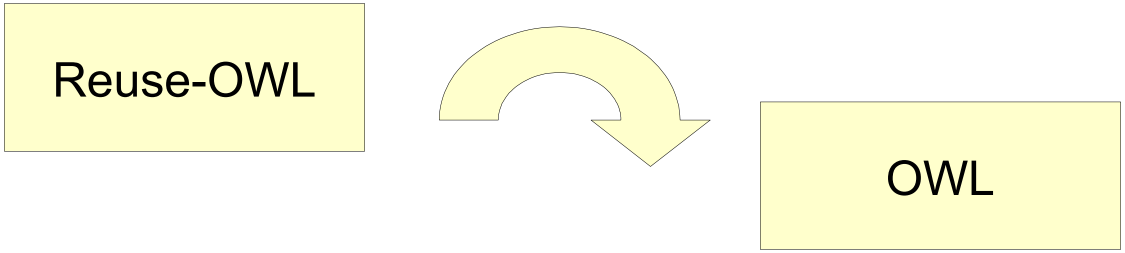


```
Ontology: http://ex.org/Faculty
  ImportRoles: http://ex.org/Board
  Class: FacultyMember
    CanPlay: BoardMember'
  Class: Professor
    SubClassOf: FacultyMember
    CanPlay: ChairMan'
  Class: PhDStudent
    SubClassOf: FacultyMember
  Individual: smith
    Types: Professor, Chairman'
  Individual: mike
    Types: PhDStudent, BoardMember'
```

```
RoleModel: http://ex.org/Board
  Role: BoardMember'
  Role: Chairman'
    SubClassOf: BoardMember' and
     electedBy' some BoardMember'
  Role: Secretary'
    SubClassOf: BoardMember'
  ObjectProperty: electedBy'
    Domain: Chairman'
    Range: BoardMember'
  ObjectProperty: appointedBy'
    Domain: Secretary'
    Range: Chairman'
```

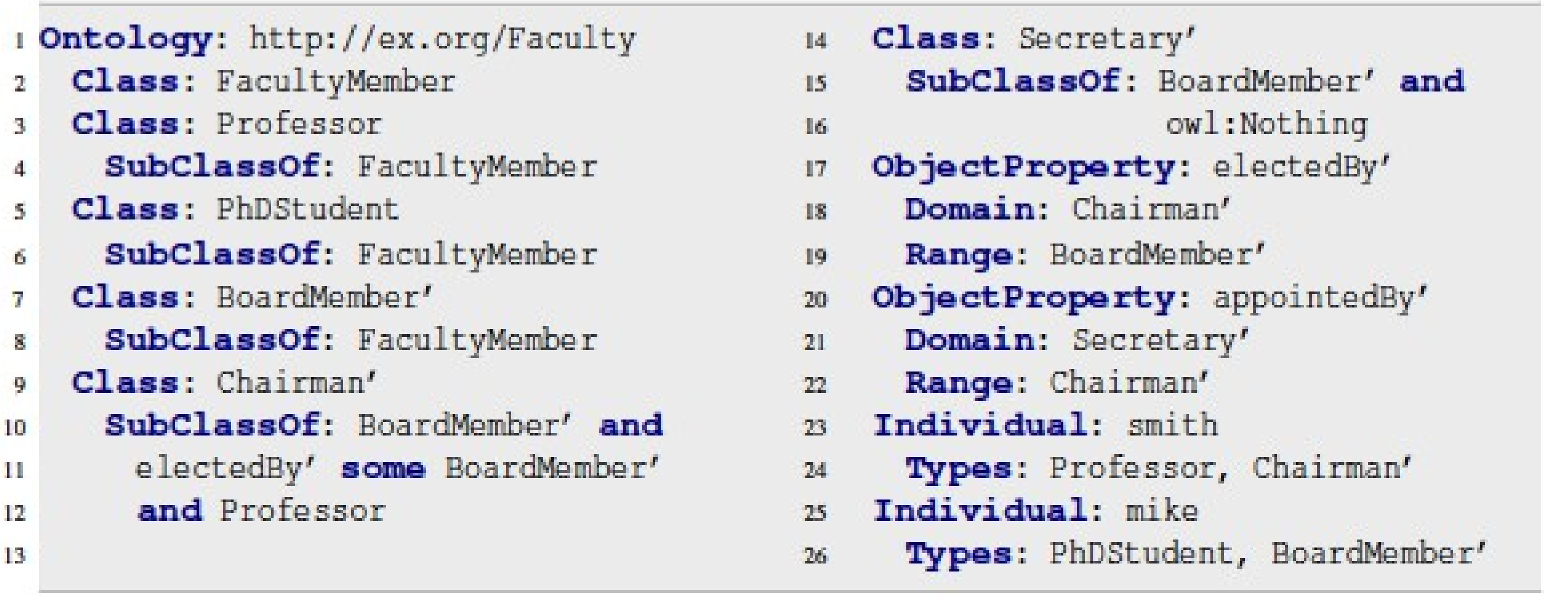
Reuse-OWL Preprocessor

- Reuse-OWL preprocessor translates to plain OWL



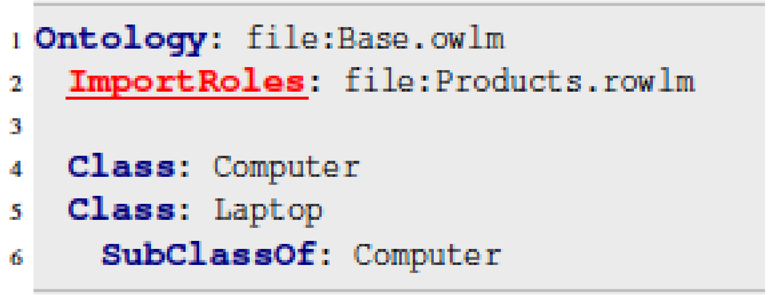
Reuse-OWL

OWL



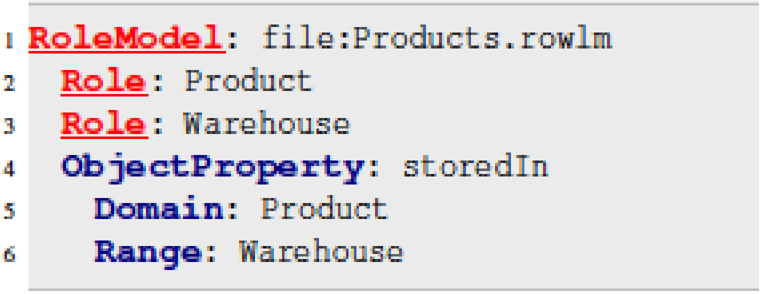
```
Ontology: http://ex.org/Faculty
  Class: FacultyMember
  Class: Professor
    SubClassOf: FacultyMember
  Class: PhDStudent
    SubClassOf: FacultyMember
  Class: BoardMember′
    SubClassOf: FacultyMember
  Class: Chairman′
    SubClassOf: BoardMember′ and
      electedBy′ some BoardMember′
      and Professor

Class: Secretary′
  SubClassOf: BoardMember′ and
                 owl:Nothing
ObjectProperty: electedBy′
  Domain: Chairman′
  Range: BoardMember′
ObjectProperty: appointedBy′
  Domain: Secretary′
  Range: Chairman′
Individual: smith
  Types: Professor, Chairman′
Individual: mike
  Types: PhDStudent, BoardMember′
```

Result of the Embedded Invasive Composition



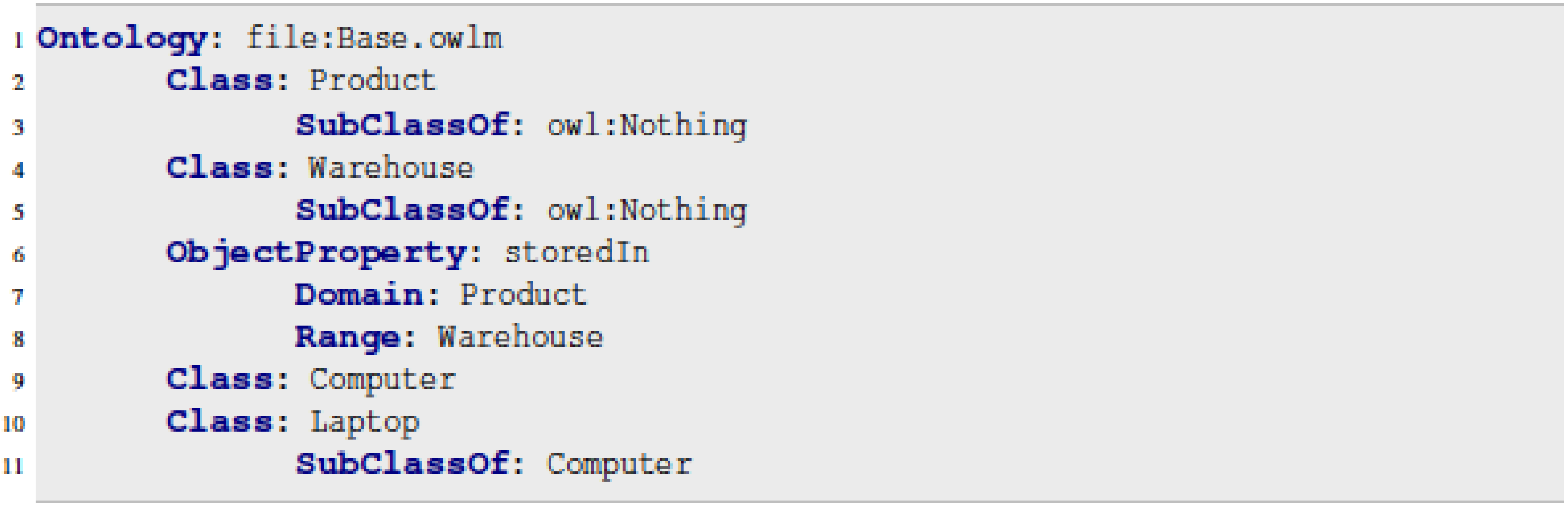
```
Ontology: file:Base.owlm
  ImportRoles: file:Products.rowlm

  Class: Computer
  Class: Laptop
    SubClassOf: Computer
```

LISTING 6.7: *Base ontology.*



```
RoleModel: file:Products.rowlm
  Role: Product
  Role: Warehouse
  ObjectProperty: storedIn
    Domain: Product
    Range: Warehouse
```

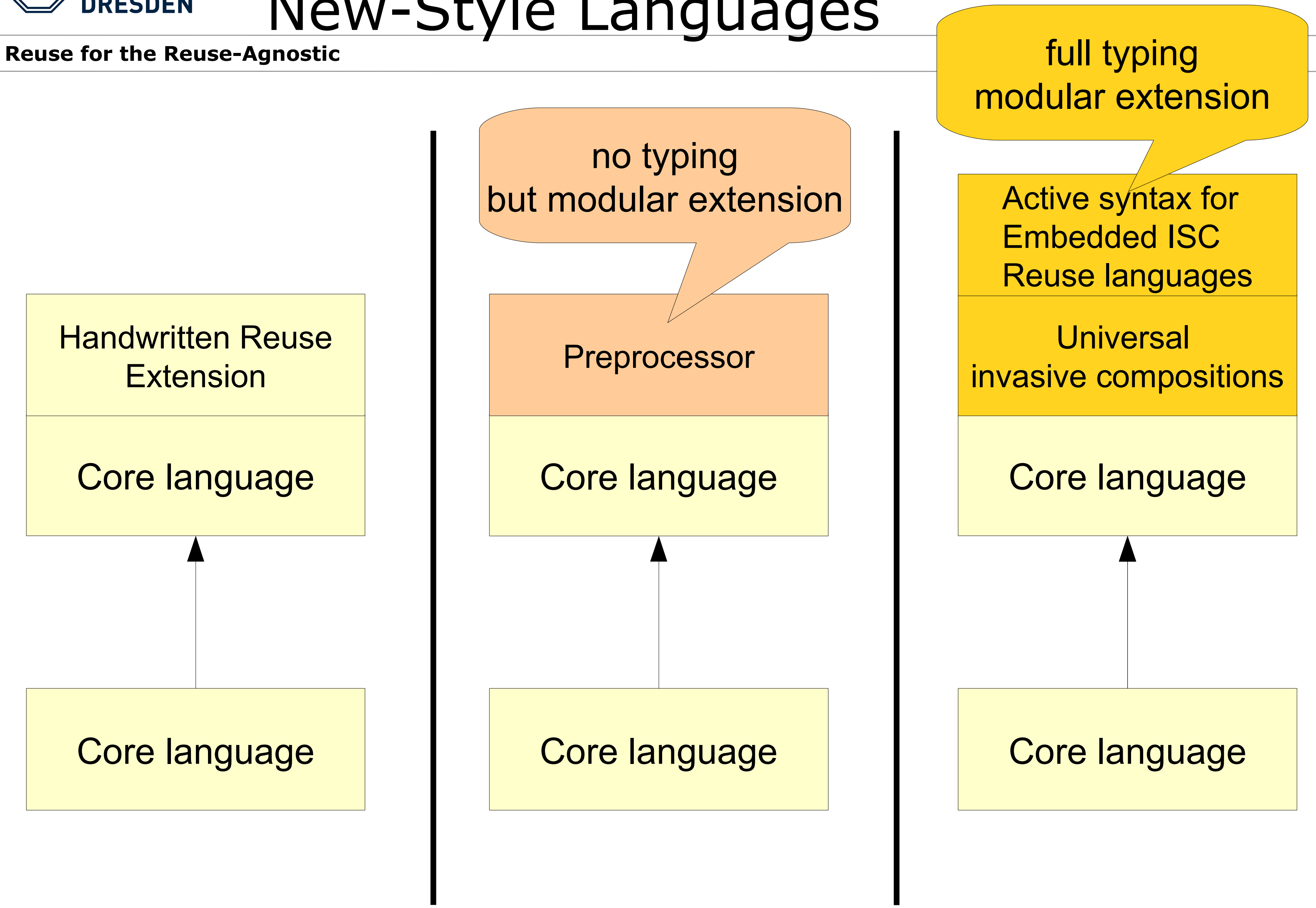
LISTING 6.8: *Role model.*



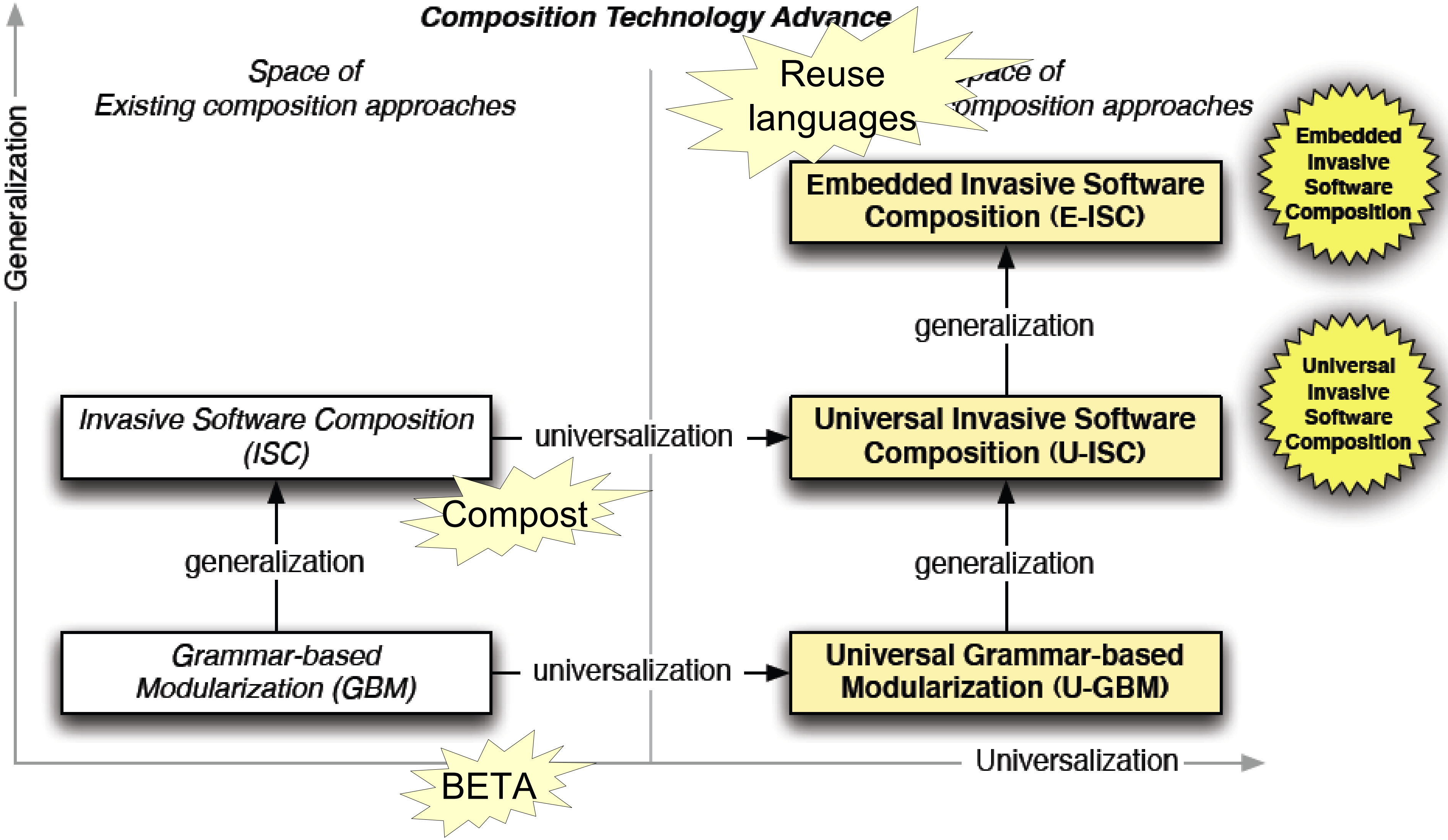
```
Ontology: file:Base.owlm
    Class: Product
        SubClassOf: owl:Nothing
    Class: Warehouse
        SubClassOf: owl:Nothing
    ObjectProperty: storedIn
        Domain: Product
        Range: Warehouse
    Class: Computer
    Class: Laptop
        SubClassOf: Computer
```

Conclusion

From Old-Style Languages to New-Style Languages



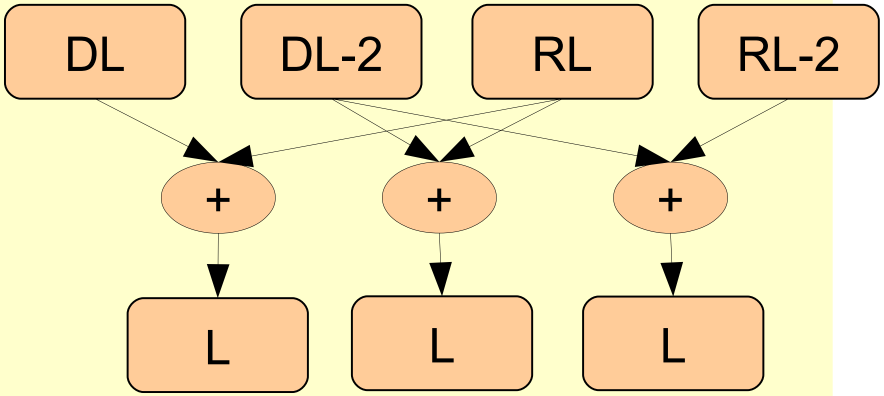
Grammar-controlled Invasive Composition


Composition Technology Advance
Generalization
Universalization
Space of
Existing composition approaches
Space of
composition approaches
Reuse languages
Invasive Software Composition (ISC)
Grammar-based Modularization (GBM)
Embedded Invasive Software Composition (E-ISC)
Universal Invasive Software Composition (U-ISC)
Universal Grammar-based Modularization (U-GBM)
generalization
universalization
Embedded Invasive Software Composition
Universal Invasive Software Composition
Compost
BETA

The Vision: Heterogenous Programs

- Reuse statements can be imported from reuse language components
- Embedded ISC offers type-safe, language specific reuse languages

```
use SQL.5.0 for query
use Modula.2.0 for scopes
use C++.2040 for class templates
use BETA for slots
template class S, DB {
  IMPLEMENTATION MODULE WebServer<S>;
    PROCEDURE <<..>> END;
  BEGIN
    S: servletGenerator = DB.init;
    R: relation = select all from DB
                       where Person == “Assmann”;
  END
}
```



Get rid of your CPP!

Replace it by a
Reuse Language Preprocessor!

Conclusion & Future work

- **Modern software development requires lots of new languages**
  - Often developed specifically for one objective
  - More technical issues—e.g., reuse—not covered
- **Reuseware provides a generic mechanism for implementing reuse and components for arbitrary languages**
  - Formalism for language extension to improve variability and extensibility
  - Mechanism for language extendsion with first-class constructs for composition
- **Future work**
  - Ensuring semantic correctness of composition
  - Defining the formalism for metamodels
  - Applying our work on the meta-level for language composition (grammar/metamodel languages)

Bierkasten Research is about Reuse Languages

http://st.inf.tu-dresden.de

http://reuseware.org

Chumwa CCBYSA-3.0 https://de.wikipedia.org/wiki/Datei:BierkistenAufPaletten.jpg

The End

- http://www.jot.fm/issues/issue_2007_10/paper7/