Modules with First-Class Transformations

Klaus Ostermann

Joint work with Sebastian Erdweg
University of Marburg, Germany

Motivation 1:
Putting build scrips in programs

- Code transformations are ubiquitous in large software systems
  - Compilers
  - Parser generators
  - Cross-language integration
  - Persistence frameworks
  - XML integration
  - Model-Driven Development
  - …

Motivation 1: Putting build scripts in programs

- Problem: Transformations are not well-integrated into module systems
  - **Lack of communication integrity** hinders separate compilation and separate understandability
  - **Lack of composability** hinders effective decompositions
  - **Lack of uniformity** hinders transformations across meta-levels and uniform applicability of transformations to all artefacts

Lack of Communication Integrity

- Ex: Module imports generated parser, dependency on grammar implicit
- Modules depend on modules they are not explicitly connected to
  - Dependencies hidden in build scripts, generator/workflow models
- Global compilation or manual tracking of dependencies required
- Breaks abstraction barrier (specification vs implementation) of code transformation

Lack of Composability

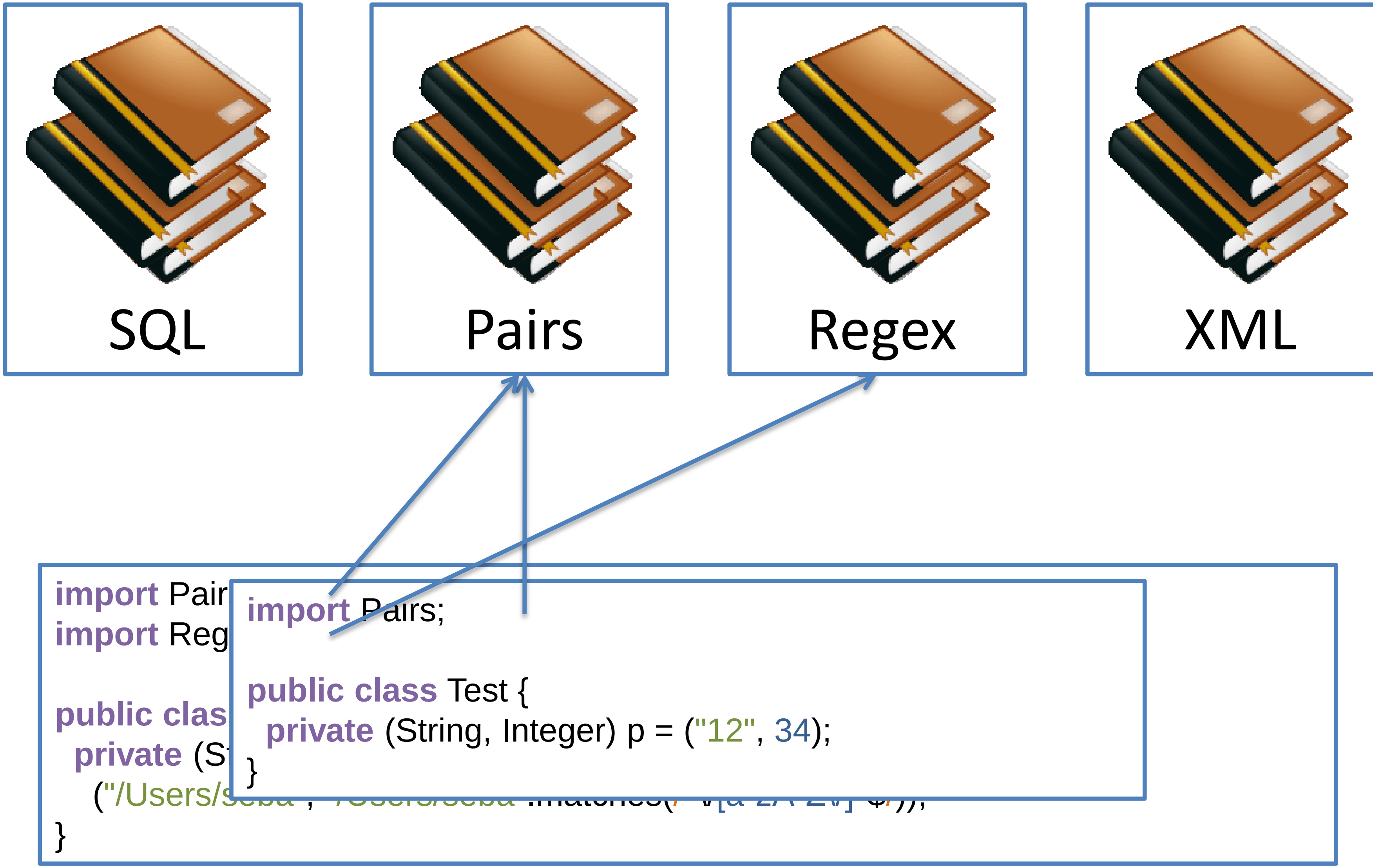
- Transformations usually coarse-grained, work on the file level
- Transformations usually do not compose
- Makes it hard to use and customize results of transformations
  - „Hacks“ such as protected regions or inheriting from generated classes
- Makes it hard to use many transformations locally (macro style)

Lack of Uniformity

- Transformations should be uniformly applicable across meta-levels
  - Higher-order transformations
  - Meta-level-polymorphic models & transformations
- Every artefact should be available for transformation
  - „Everything is a model"

Motivation 2: Improving SugarJ

- SugarJ = Syntactic extensibility by libraries for Java
- Can be considered a macro system with particularly flexible macro call syntax


SQL
Pairs
Regex
XML
import Pair
import Reg
public clas
private (St
("/Users/s
}
import Pairs;
public class Test {
private (String, Integer) p = ("12", 34);
}

Example: XML serialization

in Java using SAX

```
public void appendBook(ContentHandler ch) {
  String title = "Sweetness and Power";
  ch.startDocument();
  AttributesImpl bookAttrs = new AttributesImpl();
  bookAttrs.addAttribute("", "title", "title", "CDATA", title);
  ch.startElement("", "book", "book", bookAttrs);
  AttributesImpl authorAttrs = new AttributesImpl();
  authorAttrs.addAttribute("", "name", "name", "CDATA", "Sidney W. Mintz");
  ch.startElement("", "author", "author", authorAttrs);
  ch.endElement("", "author", "author");
  ch.startElement("", "editions", "editions", new AttributesImpl());
  AttributesImpl edition1Attrs = new AttributesImpl();
  edition1Attrs.addAttribute("", "year", "year", "CDATA", "1985");
  edition1Attrs.addAttribute("", "publisher", "publisher", "CDATA", "Viking");
  ch.startElement("", "edition", "edition", edition1Attrs);
  ch.endElement("", "edition", "edition");
  ch.endElement("", "editions", "editions");
  ch.endElement("", "book", "book");
  ch.endDocument();
}
```

XML in SugarJ

```
import XML;

public void appendBook(ContentHandler ch) {
  String title = "Sweetness and Power";

  ch.<book title="{title}">
       <author name="Sidney W. Mintz" />
       <editions>
         <edition year="1985" publisher="Viking Press" />
         <edition year="1986" publisher="Penguin Books" />
       </editions>
     </book>;
}
```

```
public sugar Pairs {

  context-free syntax
    "(" JavaExpr "," JavaExpr ")" -> JavaExpr


  rules
    pair-desugaring
      |[ (~e1, ~e2) ]|

  desugarings
    pair-desugaring
}
```

← SDF

```
import Pairs;

public class Test {
  private (String, Integer) p = ("12", 34);
}
```

← Stratego

```
private (String, Integer) p = ("12", 34);
```

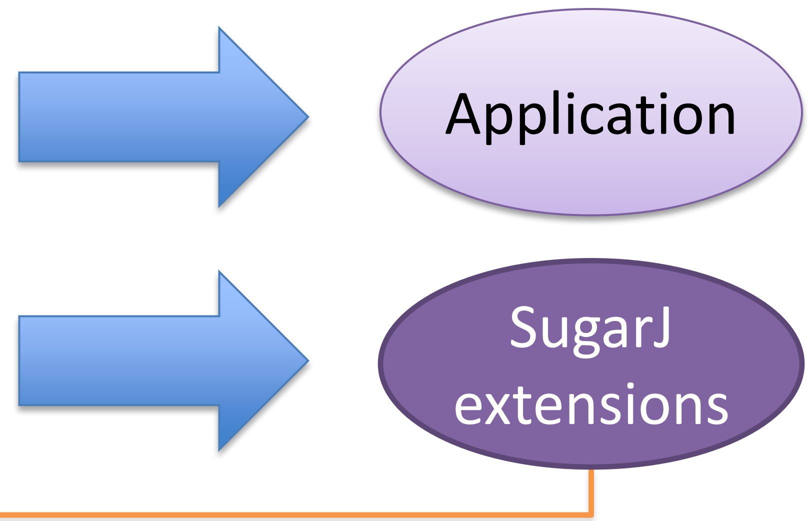
Desugar →

```
private Pair<String, Integer> p = new Pair("12", 34);
```

Metalevels and SugarJ

SugarJ is

- object language
- metalanguage



libraries can affect both

Metalanguage extensions

- alternative syntax for grammars
- concrete syntax for transformations
- met

```
rules
  pair-desugaring :
    |[ (~e1, ~e2) ]| -> |[ new Pair(~e1, ~e2) ]|
```

```
< prog   :   stat+ ;
  stat   :        expr NEWLIN
         |        ID '=' expr                    D expr)
         |        NEWLINE                         ;
```

desugar

```
rules
  pair-desugaring :
    PExpr(e1, e2) ->
    NewInstance(
      None(),
      ClassOrInterfaceType(TypeName(Id("Pair")), None()),
      [e1, e2],
      None())
```

Problem: Coupling of Syntax & Transformation

```
public sugar Pairs {

  context-free syntax
    "(" JavaExpr "," JavaExpr ")" -> JavaExpr


  rules
    pair-desugaring :
      |[ (~e1, ~e2) ]| -> |[ new Pair(~e1, ~e2) ]|

  desugarings
    pair-desugaring
}
```

- For instance, cannot store XML data independent of transformation
- Different transformations would make sense but we can only choose one

Our approach

- A module system with first-class transformations
  - Transformations and input to transformations (*models*) are independent
  - Transformations are applied on-demand in import statements
  - Can apply many transformations to the same models
  - Can reify everything as a model, including transformations
  - Supports higher-order transformations

Example: A state machine DSL

```
package banking;

import statemachine.Metamodel;

public statemachine ATM {
  initial state Init

  events DoWithdraw, Cancel, PinOK, PinNOK, [...]

  state Init {
    DoWithdraw => Withdraw
    Cancel => Init
  }
  state Withdraw {
    PinOK => GiveMoney
    PinNOK => RevokeCard
    Cancel => Init
  }
  state GiveMoney { MoneyTaken => ReturnCard }
  state ReturnCard { CardTaken => Init }
  state RevokeCard { CardRevoked => Init }
}
```

State machine metamodel

```
package statemachine;
public metamodel Metamodel {
  context-free syntax
    Statemachine -> ToplevelDeclaration
    Mod* "statemachine" Id "{" SMBody "}" -> Statemachine
    InitialState EventsDec* StateDec* -> SMBody
    ... }
```

Using the state machine

```
package banking;

import banking.ATM<statemachine.SM2Java> as ATMJ;

public class ATMTest {
  public void test() {
    ATMJ machine = new ATMJ();
    machine.step(machine.event_DoWithdraw());
    machine.step(machine.event_PinOK());
    machine.step(machine.event_MoneyTaken());
    machine.step(machine.event_CardTaken());
    assert machine.currentState() == machine.state_Init();
}}
```

Formalization (1/3)

Syntax:

$n \in \text{Name}$		
$m ::= (\overline{n} = \overline{i} \text{ in } e)$	module has imports and a body
$i ::= n \mid i\langle i \rangle$	import named, import transformed
$e ::= \ldots$	module body left abstract

Semantic domains:

$\mathbb{D} = \mathbb{M} \times (\mathbb{B} + \mathbb{T} + \{\bullet\})$	
$\mathbb{B} = \ldots$	base semantics left abstract
$\mathbb{M} = m \times \Gamma$	model closes over module's dependencies
$\mathbb{T} = \mathbb{M} \to \mathbb{D}_\perp$	transformations
$\Gamma = \text{Name} \to \mathbb{D}_\perp$	environments

Formalization (2/3)

Semantics:

$$\textit{sem-mod} \;:\; m \rightarrow \Gamma \rightarrow \mathbb{D}_{\perp}$$

$$\textit{sem-mod}(\overline{n} = \overline{i} \text{ in } e, \rho) = \begin{cases} \perp, & \text{if } \perp \in \overline{d} \\ \perp, & \text{if } \textit{body} = \perp \\ (m, \textit{body}), & \text{otherwise} \end{cases}$$

$$\begin{aligned} \text{where} \quad d_x \in \overline{d} &= \textit{resolve}(i_x, \rho) \text{ for } i_x \in \overline{i} \\ \sigma &= \textit{mkenv}(\overline{n}, \overline{d}) \\ \textit{body} &= \textit{sem-exp}(e, \sigma) \\ m &= ((\overline{n} = \overline{i} \text{ in } e), \sigma) \end{aligned}$$

$$\textit{sem-exp} \;:\; e \rightarrow \Gamma \rightarrow (\mathbb{B} + \mathbb{T} + \{\bullet, \perp\})$$

$$\textit{sem-exp}(e, \rho) = \ldots$$

Formalization (3/3)

$$\mathit{resolve} \;:\; i \to \Gamma \to \mathbb{D}_\perp$$

$$\mathit{resolve}(i, \rho) = \begin{cases} \rho(n), & \text{if } i = n \\ d_2(m_1), & \text{if } i = i_1\langle i_2 \rangle \\ & \text{and } (m_1, d_1) = \mathit{resolve}(i_1, \rho) \\ & \text{and } (m_2, d_2) = \mathit{resolve}(i_2, \rho) \\ & \text{and } d_2 \in \mathbb{T} \\ \perp, & \text{otherwise} \end{cases}$$

$$\mathit{mkenv} \;:\; \overline{n} \times \overline{\mathbb{D}} \to \Gamma$$

$$\mathit{mkenv}(\varepsilon, \varepsilon) = \lambda n.\ \perp$$

$$\mathit{mkenv}(n \cdot \overline{n}, d \cdot \overline{d}) = \lambda n'.\ \begin{cases} d, & \text{if } n = n' \\ \mathit{mkenv}(\overline{n}, \overline{d})(n'), & \text{otherwise} \end{cases}$$

Properties

Theorem 1 (Communication integrity): For all modules m and environments ρ and σ, if $\rho|_{deps\text{-}mod(m)} = \sigma|_{deps\text{-}mod(m)}$ then $sem\text{-}mod(m, \rho) = sem\text{-}mod(m, \sigma)$.

Theorem 2 (Separate compilation): For all modules m, environments ρ, and names n , if $n \notin deps\text{-}mod(m)$ then $sem\text{-}mod(m, \rho) = sem\text{-}mod(m, \rho|_{dom(\rho)\setminus\{n\}})$.

Conclusions

- We want to make transformations a first class citizen in module languages
- We try to combine the strengths of macros and MDD
- All dependencies are explicit
- Everything is (or can be) a model
- All concepts applicable across meta-levels
- Try it! Download on http://sugarj.org