Safely-Composable Type-Specific Languages (TSLs)

Cyrus Omar
Darya Kurilova
Ligia Nistor
Benjamin Chung
Alex Potanin *(Victoria University of Wellington)*
Jonathan Aldrich

School of Computer Science
Carnegie Mellon University

Specialized notations are useful.

MATHEMATICS

SPECIALIZED NOTATION

$$f(x) = \mathcal{O}(x^2)$$

GENERAL-PURPOSE NOTATION

There exists a positive constant M such that for all sufficiently large values of x, the absolute value of f(x) is at most M multiplied by x^2.

Specialized notations are useful.

DATA STRUCTURES

SPECIALIZED NOTATION

```
[1, 2, 3, 4, 5]
```

GENERAL-PURPOSE NOTATION

```
Cons(1, Cons(2, Cons(3, Cons(4, Cons(5, Nil)))))
```

Specialized notations are useful.

REGULAR EXPRESSIONS

SPECIALIZED NOTATION

```
/\d\d:\d\d\w?((a|p)\.?m\.?)/
```

GENERAL-PURPOSE NOTATION

```
Concat(Digit, Concat(Digit, Concat(Char ':', Concat(Digit, Concat(Digit,
Concat(ZeroOrMore(Whitespace), Group(Concat(Group(Or(Char 'a’,
Char 'p')), Concat(Optional(Char '.'), Concat(Char 'm',
Optional(Char '.'))))))))))))
```

STRING NOTATION

string literals have their own semantics

```
rx_from_str("\\d\\d:\\d\\d\\w?((a|p)\\.?m\\.?)")
```

parsing happens at run-time

(cf. Omar et al., ICSE 2012)

Specialized notations are useful.

QUERY LANGUAGES (SQL)

SPECIALIZED NOTATION

```
query(db, <SELECT * FROM users WHERE name={name} AND pwhash={hash(pw)}>)
```

GENERAL-PURPOSE NOTATION

```
query(db, Select(AllColumns, "users", [
  WhereClause(AndPredicate(EqualsPredicate("name", StringLit(name)),
    EqualsPredicate("pwhash", IntLit(hash(pw))))]))
```

STRING NOTATION

injection attacks

```
query(db, "SELECT * FROM users WHERE name='"+name+"' AND pwhash="+hash(pw))
```

```
'; DROP TABLE users --
```

Specialized notations are useful.

TEMPLATE LANGUAGES

SPECIALIZED NOTATION

```
<html><body><h1>Results for {keyword}</h1><ul id="results">{
  to_list_items(query(db,
    <SELECT title, snippet FROM products WHERE {keyword} in title>))}
</ul></body></html>
```

GENERAL-PURPOSE NOTATION

```
HTMLElement({}, [BodyElement({}, [H1Element({}, [Text "Results for " + keyword]),
  ULElement({id: "results"}, to_list_items(exec_query(db,
    Select(["title", "snippet"], "products", [
      WhereClause(InPredicate(StringLit(keyword), "title"))]))))]]]
```

parsing happens at run-time

cross-site scripting attacks

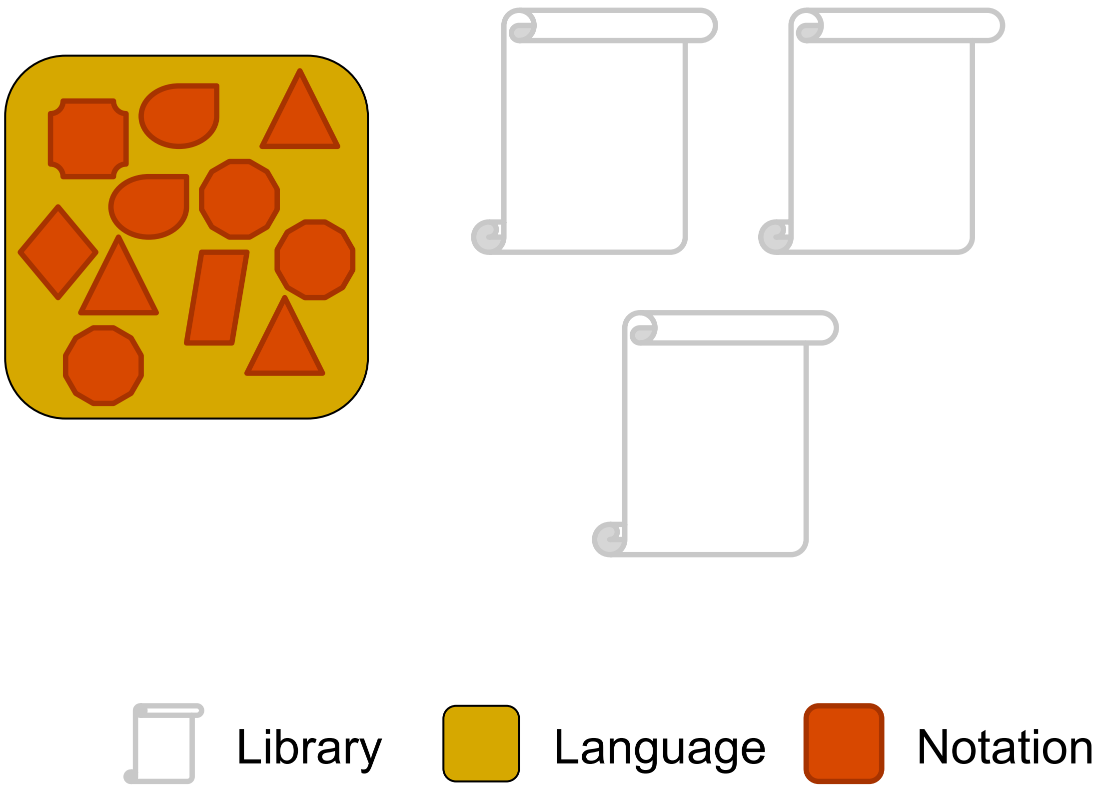
awkwardness

STRING CONCATENATION

```
html_from_str("<html><body><h1>Results for "+keyword+"</h1><ul id=\"results\">"
  + to_list_items(query(db,
      "SELECT title, snippet WHERE '"+keyword+"' in title FROM results")) +
"</ul></body></html>")
```

injection attacks

Specialized notations typically require the cooperation of the language designer.

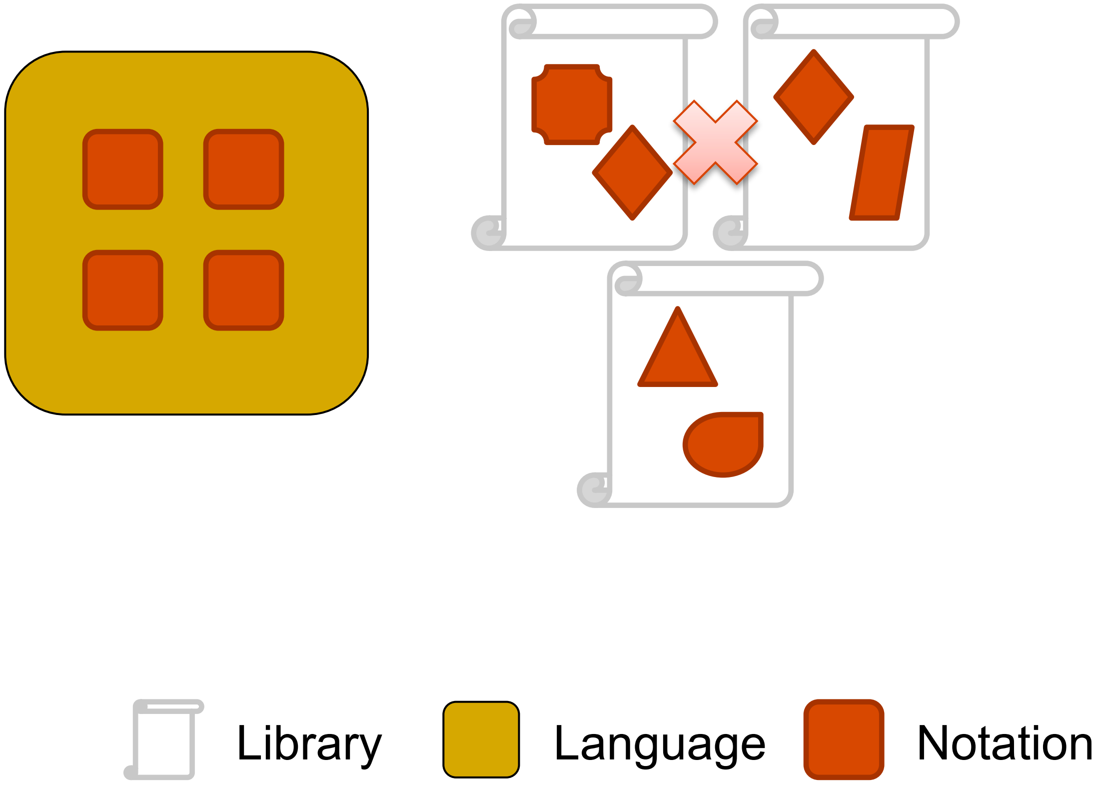


String notations are ubiquitous.

Classes in Java Corpus	Count
Total	125,048
Constructor takes a string argument	30,190
String argument is parsed	**19,317**

There are more things in heaven and earth, Horatio,
Than are dreamt of in your philosophy. *- Hamlet Act 1, scene 5*

Better approach: an extensible language where specialized notations can be distributed in libraries.



Expressivity vs. Safety

- We want to permit **expressive syntax extensions**.
- But if you give each extension too much control, they may **interfere with one another** in combination!

Example: Sugar* [Erdweg et al, 2010; 2013]

- Libraries can extend the **base syntax** of the language
- These extensions are imported **transitively**
- Extensions can **interfere**:
  - Pairs vs. *n*-tuples – what does `(1, 2)` mean?
  - HTML vs. XML – what does `<section>` mean?
  - Sets vs. Dicts – what does `{ }` mean?
  - Different *implementations* of the same abstraction

The Argument So Far

- **Specialized notations are preferable** to general-purpose notations and string notations in a variety of situations.
- It is **unsustainable for language designers** to attempt to anticipate all useful specialized notations.
- But it is also **a bad idea to give users free reign** to add arbitrary specialized notations to a base grammar.

Our Solution

- Libraries **cannot** extend the **base syntax** of the language
- Instead, **notation is associated with types**.

"Type-Specific Languages" (TSLs)

- A type-specific language can be used within **delimiters** to **create values of that type**.

"Safely-Composable"

Wyvern

- **Goals:** Secure web and mobile programming within a single statically-typed language.

- Compile-time support for a variety of **domains**:
  - Security policies and architecture specifications
  - Client-side programming (HTML, CSS)
  - Server-side programming (Databases)

Example

```
serve : (URL, HTML) -> ()
```

base language
URL TSL
HTML TSL
CSS TSL
String TSL
SQL TSL

```
serve(`products.nameless.com`, ~)
  :html
    :head
      :title Product Listing
      :style {~
        body { font-family: {bodyFont} }
      }
    :body
      :div[id="search"]
        {SearchBox("Products")}
      :ul[id="products"]
        {items_from_query(query(db,
           <SELECT * FROM products COUNT {n_products}>))}
```

- How do you **enter a TSL**?

```
base language
URL TSL
HTML TSL
CSS TSL
String TSL
SQL TSL
```

```
serve(`products.nameless.com`, ~)
  :html
    :head
      :title Product Listing
      :style {~
        body { font-family: {bodyFont} }
      }
    :body
      :div[id="search"]
        {SearchBox("Products")}
      :ul[id="products"]
        {items_from_query(query(db,
           <SELECT * FROM products COUNT {n_products}>))}
```

TSL Delimiters

- In the base language, several **inline delimiters** can be used to create a *TSL literal*:
  - ``` `TSL code here, ``inner backticks`` must be doubled` ```
  - `'TSL code here, ''inner single quotes'' must be doubled'`
  - `{TSL code here, {inner braces} must be balanced}`
  - `[TSL code here, [inner brackets] must be balanced]`
  - `<TSL code here, <inner angle brackets> must be balanced>`
- If you use the **block delimiter**, tilde (~), there are no restrictions on the subsequent *TSL literal*.
  - Indentation ("layout") determines the end of the block.
  - One block delimiter per line.

✓ How do you **enter a TSL**?

➢ How do you **associate a TSL with a type?**

```
base language
URL TSL
HTML TSL
CSS TSL
String TSL
SQL TSL
```

```
serve(`products.nameless.com`, ~)
  :html
    :head
      :title Product Listing
      :style {~
        body { font-family: {bodyFont} }
      }
    :body
      :div[id="search"]
        {SearchBox("Products")}
      :ul[id="products"]
        {items_from_query(query(db,
           <SELECT * FROM products COUNT {n_products}>))}
```

Associating a `Parser` with a type

```
casetype HTML =
    Text of String
  | DIVElement of (Attributes, HTML)
  | ULElement  of (Attributes, HTML)
  | ...
  metadata = new
    val parser : Parser = new
      def parse(s : TokenStream) : ExpAST =
        (* code to parse specialized HTML notation *)
```

```
objtype Parser =
  def parse(s : TokenStream) : ExpAST
```

```
casetype ExpAST =
  Var of ID
| Lam of Var * ExpAST | Ap of Exp * Exp
| CaseIntro of TyAST * String * ExpAST | ...
```

Associating a grammar with a type

```
casetype HTML =
    Text of String
  | DIVElement of (Attributes, HTML)
  | ULElement  of (Attributes, HTML)
  | ...
  metadata = new
    val parser : Parser = ~
      start ::= “:body” children=start => {~
                    HTML.BodyElement(([], `children`))
                }
              | ...
```

Grammars are TSLs for Parsers!

Quotations are TSLs for ASTs!

- ✓ How do you **enter a TSL**?
- ✓ How do you **associate a TSL with a type?**
- ➢ How do you **exit a TSL?**

```
base language
URL TSL
HTML TSL
CSS TSL
String TSL
SQL TSL
```

```
serve(`products.nameless.com`, ~)
  :html
    :head
      :title Product Listing
      :style {~
        body { font-family: {bodyFont} }
      }
    :body
      :div[id="search"]
        {SearchBox("Products")}
      :ul[id="products"]
        {items_from_query(query(db,
           <SELECT * FROM products COUNT {n_products}>))}
```

Exiting back to the base language

```
casetype HTML =
    Text of String
  | DIVElement of (Attributes, HTML)
  | ULElement  of (Attributes, HTML)
  | ...
  metaobject = new
    val parser : Parser = ~
      start ::= ":body" children=start => {~
                    HTML.BodyElement(([], `children`))
                 }
              | ...
              |  ":style" "{" e=EXP["}"] => {~
                    HTML.StyleElement(([], `e` : CSS))
                 }
```

- ✓ How do you **enter a TSL**?
- ✓ How do you **associate a TSL with a type?**
- ✓ How do you **exit a TSL?**
- ➢ How do **parsing and typechecking work?**

```
base language
URL TSL
HTML TSL
CSS TSL
String TSL
SQL TSL
```

```
serve(`products.nameless.com`, ~)
  :html
    :head
      :title Product Listing
      :style {~
        body { font-family: {bodyFont} }
      }
    :body
      :div[id="search"]
        {SearchBox("Products")}
      :ul[id="products"]
        {items_from_query(query(db,
           <SELECT * FROM products COUNT {n_products}>))}
```

Wyvern Abstract Syntax

$$
\begin{array}{rcl}
\rho & ::= & \textbf{objtype}\ t\ = \{\omega, \textbf{metaobject} = e\}; \rho \\
 & | & \textbf{casetype}\ t\ = \{\chi, \textbf{metaobject} = e\}; \rho \\
 & | & e \\
\\
\tau & ::= & t \\
 & | & \tau \rightarrow \tau \\
\\
e & ::= & x \\
 & | & \boldsymbol{\lambda} x{:}\tau.e \\
 & | & e(e) \\
 & | & t.C(e) \\
 & | & \textbf{case}\ e\ \textbf{of}\ \{c\} \\
 & | & \textbf{new}\ \{d\} \\
 & | & e.f \\
 & | & e.m \\
 & | & e : \tau \\
 & | & t.\textbf{metaobject} \\
 & | & \lfloor literal \rfloor
\end{array}
$$

Bidirectional Typechecking

$$\boxed{\mathit{\Delta};\mathit{\Gamma} \vdash e \uparrow \tau \rightsquigarrow \hat{e}}$$

from the type context $\mathit{\Delta}$ and the variable context $\mathit{\Gamma}$ we synthesize the type τ for e. The expression e possibly containing $\lfloor literal \rfloor$ forms is transformed into the expression $\hat{e}$ without literals.

$$\boxed{\mathit{\Delta};\mathit{\Gamma} \vdash e \downarrow \tau \rightsquigarrow \hat{e}}$$

we check e against the type τ

$$\frac{\mathit{\Delta};\mathit{\Gamma} \vdash e \uparrow \tau_1 \to \tau_2 \rightsquigarrow \hat{e} \quad \mathit{\Gamma} \vdash e_1 \downarrow \tau_1 \rightsquigarrow \hat{e}_1}{\mathit{\Delta};\mathit{\Gamma} \vdash e(e_1) \uparrow \tau_2 \rightsquigarrow \hat{e}(\hat{e}_1)} \ \textit{T-appl}$$

Bidirectional Typechecking

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash t.\mathbf{metaobject}.parser \downarrow Parser \leadsto \hat{e}_p \qquad \text{TokenStream of } \lfloor literal \rfloor \text{ is } \hat{e}_{ts} \\ \hat{e}_p.parse(\hat{e}_{ts}) \Downarrow Exp.C(\hat{e}') \qquad Exp.C(\hat{e}') \hookrightarrow e \qquad \Delta;\Gamma \vdash e \downarrow t \leadsto \hat{e}\end{array}}{\Delta;\Gamma \vdash \lfloor literal \rfloor \downarrow t \leadsto \hat{e}} \ \text{T-LITERAL}$$

- ✓ How do you **enter a TSL**?
- ✓ How do you **associate a TSL with a type?**
- ✓ How do you **exit a TSL?**
- ✓ How do **parsing and typechecking work?**

```
base language
URL TSL
HTML TSL
CSS TSL
String TSL
SQL TSL
```

```
serve(`products.nameless.com`, ~)
  :html
    :head
      :title Product Listing
      :style {~
        body { font-family: {bodyFont} }
      }
    :body
      :div[id="search"]
        {SearchBox("Products")}
      :ul[id="products"]
        {items_from_query(query(db,
           <SELECT * FROM products COUNT {n_products}>))}
```

Benefits

- **Modularity and <u>Safe</u> Composability**
  - DSLs are distributed in libraries, along with types
  - No link-time errors possible
- **Identifiability**
  - Can easily see when a DSL is being used
  - Can determine which DSL is being used by identifying expected type
  - DSLs always generate a value of the corresponding type
- **Simplicity**
  - Single mechanism that can be described in a few sentences
  - Specify a grammar in a natural manner within the type
- **Flexibility**
  - A large number of literal forms can be seen as type-specific languages
  - Whitespace-delimited blocks can contain *arbitrary* syntax

Types Organize Languages

- Types represent an organizational unit for programming language semantics.
- Types are not only useful for traditional verification, but also **safely-composable language-internal syntax extensions.**

Limitations

- **Decidability of Compilation**
  - Because user-defined code is being evaluated during parsing and typechecking, compilation might not terminate.
  - There is work on termination analyses for attribute grammars (Krishnan and Van Wyk, SLE 2012)
  - Even projects like CompCert don't place a huge emphasis on termination of parsing and typechecking.
- **No story yet for editor support.**
- **Too much freedom a bad thing?**

The Argument
For a New Human-Parser Interaction

- **Specialized notations are preferable** to general-purpose notations and string notations in a variety of situations.

- It is **unsustainable for language designers** to attempt to anticipate all useful specialized notations.

- But it is also **a bad idea to give users free reign** to add arbitrary specialized notations to a base grammar.

- **Associating syntax extensions with types** is a principled, practical approach to this problem with minor drawbacks.