Composable DSLs

Jeremy G. Siek and Erik Silkensen

IFIP WG 2.11, September 2011
Our World of Interacting DSLs

Sets

Regular Expressions

SQL

Yacc

HTML

Matrix Algebra

Application
## External vs. Internal DSLs

<table>
<thead>
<tr>
<th></th>
<th>External DSLs</th>
<th>Internal DSLs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Natural Syntax</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Friendly Diagnostics</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Interoperability</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Ease of Implementation</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

We’d like to have the best of both worlds.
Want to Enable Fine-Grained Mixing of Languages

```
<html>
<head>
OPEN 'mydb';
CREATE TABLE IF NOT EXISTS LOGS (id unique, log);
INSERT INTO LOGS (id, log) VALUES (1, "foobar");
INSERT INTO LOGS (id, log) VALUES (2, "logmsg");
</head>
<body>
var results = SELECT * FROM LOGS;
var len = results.rows.length, i;
<p>Found rows: len </p>
for (i = 0; i < len; i++) {
    <p><b>results.rows.item(i).log</b></p>
}
</body>
</html>
```
We “just” need our general purpose languages to provide extensible syntax.
The Problem

Matrix Algebra

expr ::= expr "+" expr [1]
expr ::= expr "*" expr [2]
expr ::= "l" expr "l"
expr ::= expr "^T"
...

Regular Expressions

expr ::= expr "+"
expr ::= expr "*"
expr ::= expr "l" expr
...

Ambiguous Grammar!

Application

Jeremy G. Siek and Erik Silkensen
Composable DSLs
Observation

Different DSLs define different types.

- *Matrix* and *Vector* in Matrix Algebra.
- *Regex* in Regular Expressions.
- *Query* in SQL.

Systems like Isabelle [6] and MetaBorg [3, 2] type check the resulting parse forest, discarding ill-typed parse trees.

This resolves the ambiguities, but not the cost in time!
Our Approach

- Unify type checking and parsing to prune ill-typed parse trees before they get a chance to grow [1].
- type ≈ non-terminal [10, 9]
- ground rules regarding variables and scoping:
  - variables declared (and assigned a type) before use
  - curly braces are for scoping
- Our algorithm is based on Island Parsing [11], a bidirectional form of Chart Parsing.
Example: Vector and Matrix DSLs

```plaintext
grammar VectorDSL
Exp ::= Vector;
Vector ::= Vector "*" Float [left]
  | Float "*" Vector [left]
  | "(" Vector ")";
{

grammar MatrixDSL
Exp ::= Matrix;
Matrix ::= Matrix "*" Matrix [left]
  | Matrix "*" Float [left]
  | Float "*" Matrix [left];
  | Matrix "*" Vector [left]
  | "(" Matrix ")";
{
  // An application of both DSLs
  declare A:Matrix, x:Vector, a:Float;
  A * (x * a);
}  
```
Example: giving syntax to Typed Racket

```
grammar Let
Int = Integer;
Bool = Boolean;
Int ::= \#rx"^[0-9]+$";
Id ::= \#rx"^[a-zA-Z][a-zA-Z0-9]*$";
Int ::= x: Int "+" y: Int [left 1] = (+ x y);
Int ::= x: Int "*" y: Int [left 2] = (* x y);
Bool ::= x: Int "<" y: Int = (< x y);
forall T. T ::= "if" test: Bool "then" e1: T "else" e2: T =>
  (if test e1 e2);
forall T1 T2. T2 ::= "let" x:Id "=" y: T1 "" x: T1; z: T2 "" =>
  (let: ([x : T1 y]) z);
{
  let n = 7 {
    if n < 3 then 6 else 2 + n * 5 + 5
  }
}
```

The integration of grammar rules and function/macro definitions is inspired by the Lithe language [10].
Chart Parsing

- Parses for sub-strings are stored in a table. (Think memoization or dynamic programming.)
- Each cell contains edges, grammar rules with dots to mark how much is matched, and a parse tree for each edge (not shown).
Chart Parsing as an Abstract Machine

\[ A, B \] symbols (terminals and non-terminals)
\[ \alpha, \beta, \delta, \gamma \] lists of symbols
\[ A \to \alpha \] rule
\[ G \] grammar = set of rules
\[ A \to \alpha.\beta \] edge
\[ c \] chart = \( \mathbb{N} \times \mathbb{N} \to \text{edge set} \)

\[
G \vdash c \rightleftharpoons c
\]

**FUND**

\[
\frac{(A \to \alpha.B\beta) \in c(i, j) \quad B \to \gamma. \in c(j, k)}{G \vdash c \rightleftharpoons c[(i, k) \mapsto c(i, k) \cup \{A \to \alphaB.\beta\}]} \]

**BU-PRED**

\[
\frac{B \to \gamma. \in c(i, j) \quad A \to B\beta \in G}{G \vdash c \rightleftharpoons c[(i, j) \mapsto c(i, j) \cup \{A \to B.\beta\}]} \]
Our Variation on Island Parsing

- Begin by recognizing all 1-token parses of variables and constants (but not other terminals) anywhere in the input.
- Grow these saplings outwards.
- An edge now has two dots, to mark the left and right ends of the match.
Island Parsing as an Abstract Machine

\[ A \rightarrow \alpha.\beta.\gamma \quad \text{edge} \]

**Left-Fund**

\[
G \vdash c \mapsto c[(i, k) \mapsto c(i, k) \cup \{ A \rightarrow \alpha.B.\beta.\delta \}]
\]

\[
B \rightarrow .\gamma. \in c(i, j) \quad (A \rightarrow \alpha.B.\beta.\delta) \in c(j, k)
\]

**Right-Fund**

\[
G \vdash c \mapsto c[(i, k) \mapsto c(i, k) \cup \{ A \rightarrow \alpha.\beta.B.\delta \}]
\]

\[
(A \rightarrow \alpha.\beta.B\delta) \in c(i, j) \quad B \rightarrow .\gamma. \in c(j, k)
\]

**BU-Pred**

\[
G \vdash c \mapsto c[(i, j) \mapsto c(i, j) \cup \{ A \rightarrow \alpha.B.\beta \}]
\]

\[
B \rightarrow .\gamma. \in c(i, j) \quad A \rightarrow \alpha.B\beta \in G
\]
We handle associativity and precedence.

Grammar rules may be parameterized (e.g. for "if" and "let").

Analogous to function overloading, we provide grammar rule overloading; more-specific rules take precedence over less-specific rules.

We handle binding forms by parsing in multiple passes, where earlier passes skip regions between curly braces, waiting until a later pass when all the in-scope variables are known.
**Evaluation**

**Composition Scaling** Parse time with respect to the number of grammars.

**Program Scaling** Parse time with respect to program size.
We start with the following program using a vector arithmetic DSL.

We then add in more grammars, one at a time, but otherwise hold the program constant.

```plaintext
grammar G1
    S ::= Exp | Exp S;
    Vector ::= Vector "+" Vector [left];
    Exp ::= Vector;
{
    declare A:Vector;
    A + A + A + A + A + A + A + A + A + A + A + A + A
}
```
Experiment: Composition Scaling

Skipping ahead, here’s the grammar rules for sets, regular expressions and primitive integers.

```plaintext
grammar All
    S ::= Exp | Exp S;
    Vector ::= Vector "+" Vector [left]; // G1
    Set ::= Set "+" Set [left]; // G2
    Regex ::= Regex "+"; // G3
    Int ::= "+" Int; // G4
    Exp ::= Vector | Set | Regex;
{
    declare A:Vector;
    A + A + A + A + A + A + A + A + A + A + A + A + A + A
}
```
The following is the grammar we use with SGLR.

classic syntax

Exp -> S
Exp S -> S
Name -> Exp
Exp "+" Exp -> Exp // G1
Exp "+" Exp -> Exp // G2
Exp "+" -> Exp // G3
"+" Exp -> Exp // G4
Experiment: Composition Scaling

Times are averages of 5 trials, as measured by the Unix time utility.
Our back-of-the-envelope calculation gives us $O(n^2)$ time complexity for grammars that are unambiguous, taking types into account.

To experimentally check this, the next slide shows parse times for a series of programs with growing numbers of matrix multiply expressions.

The usual time complexity for Chart Parsers is $O(n^2)$ for unambiguous grammars, not taking types into account.
**Experiment: Program Scaling**

![Graph showing parse time vs. number of A terms for MetaBorg, ambiguous and type-based disambiguation.]

Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. 

Higher-Order Mixfix Syntax for Representing Mathematical Notation and its Parsing. 

The scp parsing algorithm : computational framework and formal properties. 
In Procesamiento del lenguaje natural, number 23, 1998.
*Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*.  

A general and practical approach to concrete syntax objects within ML.  

Parsing distfix operators.  

Grammatical framework.  
Lithe: a language combining a flexible syntax and classes.

Island parsing and bidirectional charts.

*Parsing Mixfix Expressions*.
Conclusion

We’ve demonstrated an approach to parsing the composition of DSLs that is resilient to syntactic ambiguity.

Future work:

- From prototype to production.
- Can we get the time complexity down to linear?
- Performance tuning to get the constant factors down.
- Integration with constrained generics a la “concepts”.
[5, 8, 10, 11, 7, 4, 1, 12]