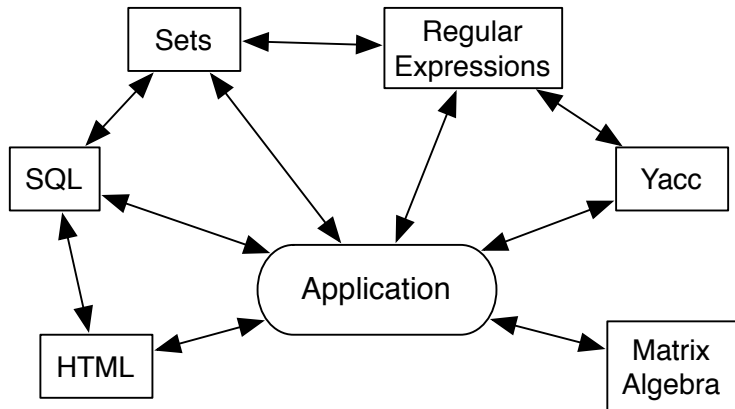


Composable DSLs

Jeremy G. Siek and Erik Silkenen

IFIP WG 2.11, September 2011

Our World of Interacting DSLs



External vs. Internal DSLs

	External DSLs	Internal DSLs
Natural Syntax	✓	
Friendly Diagnostics	✓	
Interoperability		✓
Ease of Implementation		✓

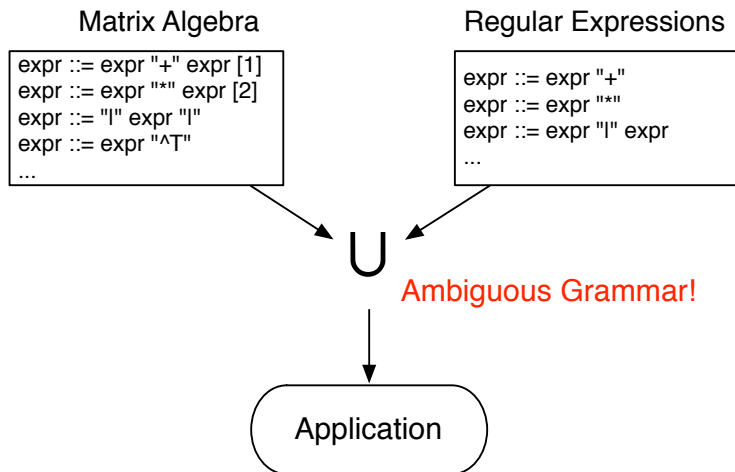
We'd like to have the best of both worlds.

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



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].
- ▶ $\text{type} \approx \text{non-terminal}$ [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

```
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).

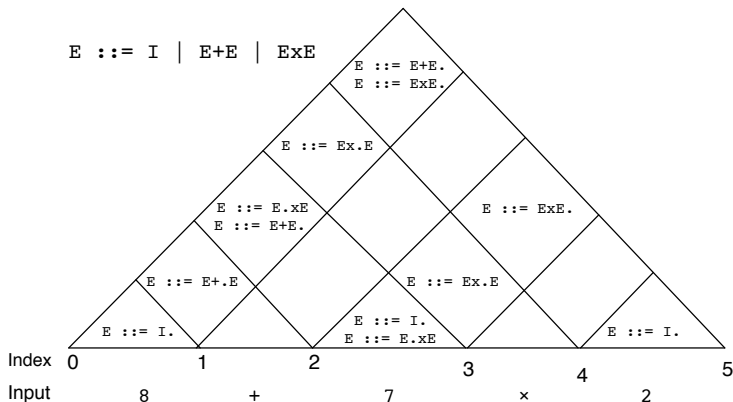


Chart Parsing as an Abstract Machine

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

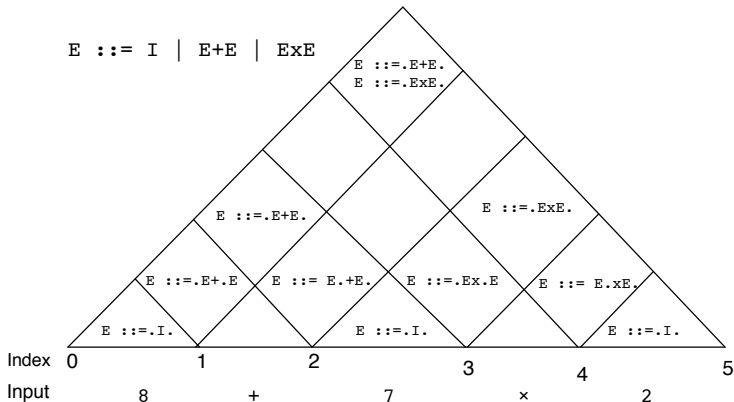
$$\boxed{G \vdash c \mapsto c}$$

$$\text{FUND} \frac{(A \rightarrow \alpha.B\beta) \in c(i, j) \quad B \rightarrow \gamma. \in c(j, k)}{G \vdash c \mapsto c[(i, k) \mapsto c(i, k) \cup \{A \rightarrow \alpha B.\beta\}]}$$

$$\text{BU-PRED} \frac{B \rightarrow \gamma. \in c(i, j) \quad A \rightarrow B\beta \in G}{G \vdash c \mapsto c[(i, j) \mapsto c(i, j) \cup \{A \rightarrow B.\beta\}]}$$

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

$A \rightarrow \alpha.\beta.\gamma$ edge

$$\text{LEFT-FUND} \frac{B \rightarrow .\gamma. \in c(i, j) \quad (A \rightarrow \alpha B.\beta.\delta) \in c(j, k)}{G \vdash c \mapsto c[(i, k) \mapsto c(i, k) \cup \{A \rightarrow \alpha.B\beta.\delta\}]}$$

$$\text{RIGHT-FUND} \frac{(A \rightarrow \alpha.\beta.B\delta) \in c(i, j) \quad B \rightarrow .\gamma. \in c(j, k)}{G \vdash c \mapsto c[(i, k) \mapsto c(i, k) \cup \{A \rightarrow \alpha.\beta B.\delta\}]}$$

$$\text{BU-PRED}^* \frac{B \rightarrow .\gamma. \in c(i, j) \quad A \rightarrow \alpha B\beta \in G}{G \vdash c \mapsto c[(i, j) \mapsto c(i, j) \cup \{A \rightarrow \alpha.B.\beta\}]}$$

- ▶ 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.

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

Program Scaling Parse time with respect to program size.

Experiment: Composition Scaling

- ▶ 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.

```
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.

```
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
}
```

The following is the grammar we use with SGLR.

context-free syntax

Exp \rightarrow S

Exp S \rightarrow S

Name \rightarrow Exp

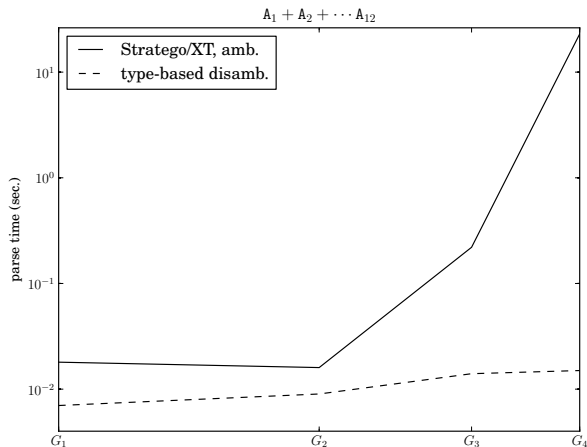
Exp "+" Exp \rightarrow Exp // G1

Exp "+" Exp \rightarrow Exp // G2

Exp "+" \rightarrow Exp // G3

"+" Exp \rightarrow Exp // G4

Experiment: Composition Scaling

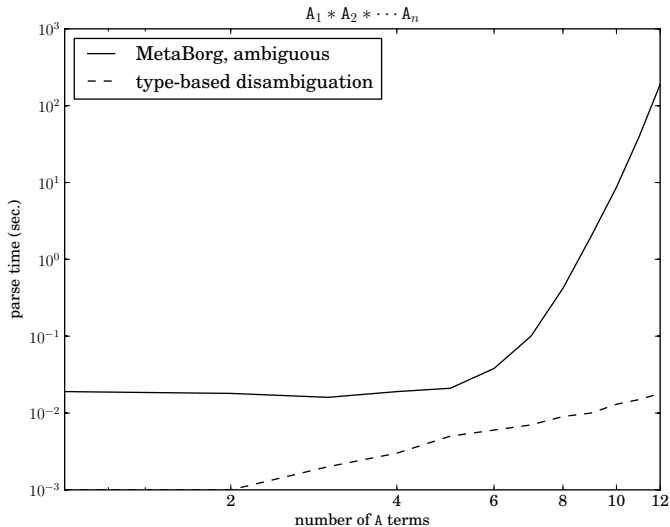


Times are averages of 5 trials, as measured by the Unix time utility.

Experiment: Program Scaling

- ▶ 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



- [1] A. Aasa, K. Petersson, and D. Synek.
Concrete syntax for data objects in functional languages.
In *Proceedings of the 1988 ACM conference on LISP and functional programming*, LFP '88, pages 96–105, New York, NY, USA, 1988. ACM.
- [2] M. Bravenboer, R. Vermaas, J. J. Vinju, and E. Visser.
Generalized type-based disambiguation of meta programs with concrete object syntax.
In R. Glück and M. R. Lowry, editors, *Generative Programming and Component Engineering, 4th International Conference, GPCE 2005*, volume 3676 of *Lecture Notes in Computer Science*, pages 157–172, Tallinn, Estonia, 2005. Springer.

- [3] M. Bravenboer and E. Visser.
Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions.
In Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, OOPSLA '04, pages 365–383, New York, NY, USA, 2004. ACM.
- [4] S. A. Missura.
Higher-Order Mixfix Syntax for Representing Mathematical Notation and its Parsing.
PhD thesis, ETH Zurich, 1997.
- [5] Q. Moreno and J. Francisco.
The scp parsing algorithm : computational framework and formal properties.
In Procesamiento del lenguaje natural, number 23, 1998.

- [6] T. Nipkow, L. C. Paulson, and M. Wenzel.
Isabelle/HOL — A Proof Assistant for Higher-Order Logic, volume 2283 of *LNCS*.
Springer, November 2007.

- [7] M. Pettersson and P. Fritzson.
A general and practical approach to concrete syntax objects within ML.
In *ACM SIGPLAN Workshop on ML and its Applications*, June 1992.

- [8] S. Peyton Jones.
Parsing distfix operators.
Commun. ACM, 29(2), February 1986.

- [9] A. Ranta.
Grammatical framework.
Journal of Functional Programming, 14(2):145–189, 2004.

- [10] D. Sandberg.
Lithe: a language combining a flexible syntax and classes.
In Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '82, pages 142–145, New York, NY, USA, 1982. ACM.
- [11] O. Stock, R. Falcone, and P. Insinnamo.
Island parsing and bidirectional charts.
In Proceedings of the 12th conference on Computational linguistics, pages 636–641, Morristown, NJ, USA, 1988. Association for Computational Linguistics.
- [12] J. Wieland.
Parsing Mixfix Expressions.
PhD thesis, Technische Universitat Berlin, 2009.

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]