Modularly Programmable Syntax

Cyrus Omar
Jonathan Aldrich

Computer Science Department
Carnegie Mellon University

[CMU POP Retreat 2015]
List Syntax

DERIVED FORM

[1, 2, 3, 4, 5]

EXPANSION

Cons(1, Cons(2, Cons(3, Cons(4, Cons(5, Nil))))))
HTML Syntax

DERIVED FORM

<h1>Welcome back, <span>name</span></h1>

EXPANSION

H1Element(Empty, Seq(
   TextNode("Welcome back, "), TextNode(name)))
Regular Expression Syntax

**DERIVED FORM**

/A|T|G|C/

**EXPANSION**

0r(0r("A", 0r("T", 0r("G", "C"))))
Many Examples

- Lists, sets, maps, multisets, vectors, arrays, …
- Monadic commands (ala Haskell `do` notation)
- Regular expressions, SQL, other database query languages
- HTML, CSS, SASS, LESS, JSON, …
- Dates, times, URLs, paths, …
- Quasiquotation, object language syntax
- Grammars
- Mathematical and scientific notations (e.g. SMILE)
Large Languages
A Better Approach: Programmable Syntax

Diagram:
- Library
- Language
- Derived Forms
import rx, html, json, kdb, list, xml

let x = /A|T|G|C/

fun greet(name : string) => <h1>Hello, <%name%></h1>
import rx, html, json, kdb, list, xml

let x = /A|T|G|C/
fun greet(name : string) => <h1>Hello, <%=name%></h1>

Composability: Can there be conflicts? With the base language? Between extensions?
import rx, html, json, kdb, list, xml

let x = /A|T|G|C/
fun greet(name : string) => <h1>Hello, <%name></h1>
let q = {(!R)&{&/x/::2_!x}'!R}

Composability: Can there be conflicts? With the base language? Between extensions?

Identifiability: Where did this syntax come from?
import rx, html, json, kdb, list, xml

let x = /A|T|G|C/  
fun greet(name : string) => <h1>Hello, <%(name)></h1>  
let q = {(!R)&&{/x!/:2_!x}'!R}
import rx, html, json, kdb, list, xml

let x = /A|T|G|C/
fun greet(name : string) => <h1>Hello, {{name}}</h1>
let q = {(!R)&{"&/x!/:2_!x'}!R}
Our Solution: Typed Syntax Macros (TSMs)

import rx, html, json, kdb, list, xml

let x = rx.$regex /A|T|G|C/
fun greet(name : string) => html.$html <h1>Hello, <%=name%></h1>
let q = kdb.$query {(!R)@&{%/x/\:2_!x}%}''!R}

Composability: Can there be conflicts? With the base language? Between extensions?

Identifiability: Where did this syntax come from?

Type Discipline: What type does q have?

Hygiene: Can I rename other variables safely? (or does the expansion capture specific variables?)
import rx, html, json, kdb, list, xml

let x = rx.$regex /A|T|G|C/
fun greet(name : string) => html.$html <h1>Hello, <span name></h1>
let q = kdb.$query {(!R)@&{%/x/:2_/!x}"!R}

Our Solution: Typed Syntax Macros (TSMs)

A fixed set of available outer delimiters prevent conflict.

Composability: Can there be conflicts? With the base language? Between extensions?

Identifiability: Where did this syntax come from?

Type Discipline: What type does q have?

Hygiene: Can I rename other variables safely? (or does the expansion capture specific variables?)
Our Solution: Typed Syntax Macros (TSMs)

```plaintext
import rx, html, json, kdb, list, xml

let x = rx.$regex /A|T|G|C/
fun greet(name : string) => html.$html <h1>Hello, <%= name%></h1>
let q = kdb.$query {(!R)@&{%/x/:2_x}''!R}
```

### Composability: Can there be conflicts?
With the base language? Between extensions?

### Identifiability: Where did this syntax come from?

### Type Discipline: What type does q have?

### Hygiene: Can I rename other variables safely?
(or does the expansion capture specific variables?)
Our Solution: Typed Syntax Macros (TSMs)

```plaintext
syntax $query at Query {
    static fn(body : Body) : Exp option => (* ... query parser here ... *)
}

import rx, html, json, kdb, list, xml

let x = rx.$regex /A|T|G|C/
fun greet(name : string) => html.$html <h1>Hello, <%name></h1>
let q = kdb.$query {(!R)@&{"/x!/:2_!x}'!R}
```

**Composability:** Can there be **conflicts**? With the base language? Between extensions?

**Identifiability:** Where did this syntax come from?

**Type Discipline:** What type does q have?

**Hygiene:** Can I rename other variables safely? (or does the expansion capture specific variables?)
import rx, html, json, kdb, list, xml

let x = rx.$regex /A|T|G|C/
fun greet(name : string) => html.$html <h1>Hello, <%=name%></h1>
let q = kdb.$query {(!R)&{&/x!/:2_!x}'!R}

Our Solution: Typed Syntax Macros (TSMs)

Composability: Can there be conflicts? With the base language? Between extensions?

Identifiability: Where did this syntax come from?

Type Discipline: What type does q have?

Hygiene: Can I rename other variables safely? (or does the expansion capture specific variables?)
import rx, html, json, kdb, list, xml

let x = rx.$regex /A|T|G|C/
fun greet(name : string) => html.$html <h1>Hello, <%=name%></h1>
let q = kdb.$query {(!R)@&{&/x!/:2_!x}'!R}

Our Solution: Typed Syntax Macros (TSMs)

Composability: Can there be conflicts? With the base language? Between extensions?

Identifiability: Where did this syntax come from?

Type Discipline: What type does q have?

Hygiene: Can I rename other variables safely? (or does the expansion capture specific variables?)
Our Solution: Typed Syntax Macros (TSMs)

\[ \Delta \Gamma \vdash \hat{e} \rightsquigarrow e : \tau \]
Pattern TSMs

```
import rx, html, json, kdb, list, xml

fun get_name(x : HTML) => match x with
    html.$html <h1>Hello, <%name></h1> => name
| _ => raise Error
```

```
pattern syntax $html at HTML {
    static fn(body : Body) : Pat option => (* ... HTML pattern parser ... *)
}
```
import rx, html, json, kdb, list, xml

fun get_name(x : HTML) => match x with
    html.$html <h1>Hello, <%(name)</h1> => name
| _ => raise Error

Pattern TSMs

**pattern syntax** $html at HTML {
  static fn(body : Body) : Pat option => (* ... HTML pattern parser ... *)
}

Only spliced subpatterns can bind variables.
The ML Module System

Abstract Datatypes

```
signature RX = sig
  type t
  val Empty : t
  val Str : string -> t
  val Seq : t * t -> t
  val Or : t * t -> t
  val Star : t -> t
  val case : (t -> 'a -> (string -> 'a) -> (t * t -> 'a) -> (t * t -> 'a) -> (t -> 'a) -> 'a)
end
```
The ML Module System

Abstract Datatypes

```ml
signature RX = sig
  type t
  val Empty : t
  val Str : string -> t
  val Seq : t * t -> t
  val Or : t * t -> t
  val Star : t -> t
  val case : (t -> 'a -> (string -> 'a) -> (t * t -> 'a) -> (t * t -> 'a) -> (t -> 'a) -> 'a)
end
```

```ml
structure Rx1 := RX = struct
  type t = string
  (* ... *)
end
```

```ml
structure Rx2 := RX = struct
  type t = Empty | Str of string
    | Seq of t * t | Or of t * t
    | Star of t
  (* ... *)
end
```

...
The ML Module System

Abstract Datatypes

signature RX = sig
  type t
  val Empty : t
  val Str : string -> t
  val Seq : t * t -> t
  val Or : t * t -> t
  val Star : t -> t
  val case : (t -> 'a ->
            (string -> 'a) ->
            (t * t -> 'a) ->
            (t * t -> 'a) ->
            (t -> 'a) ->
            'a)
end

structure Rx1 :> RX = struct
  type t = string
  (* ... *)
end

structure Rx2 :> RX = struct
  type t = Empty | Str of string
            | Seq of t * t | Or of t * t
            | Star of t
  (* ... *)
end

...
Our Solution: Parameterized TSMs

Definition

```latex
syntax $rx(R : RX) at R.t {
  static fn(body : Body) : Exp option => (* ... rx parser here ... *)
}
```

Usage

```latex
let base = $rx Rx1 /A|T|G|C/
let renzyme = $rx Rx1 /GC%{base}GC/
```
Our Solution: Parameterized TSMs

Definition

```plaintext
syntax $rx(R : RX) at R.t {
  static fn(body : Body) : Exp option => (* ... rx parser here ... *)
}
```

Usage

```plaintext
let base = $rx Rx1 /A|T|G|C/
let renzyme = $rx Rx1 /GC%{base}GC/
```
Our Solution: Type-Specific Languages (TSLs)

Definition

```plaintext
structure Rx1 => RX = struct
  type t = string
  (* ... *)
end with syntax $rx
```
Our Solution: Type-Specific Languages (TSLs)

Definition

```plaintext
structure Rx1 :> RX = struct
  type t = string
(* ... *)
end with syntax $rx
```

The TSM $rx(Rx1)$ is now associated with abstract type Rx1.t.
Our Solution: Type-Specific Languages (TSLs)

Definition

```plaintext
structure Rx1 :> RX = struct
  type t = string
  (* ... *)
end with syntax $rx
```

The TSM $rx(Rx1)$ is now associated with abstract type Rx1.t.

Usage

```plaintext
let base : Rx1.t = /A|T|G|C/
let renzyme : Rx1.t = /GC%{base}GC/
```

When a literal form appears by itself, the TSM associated with the type it is being checked against is applied.
Our Solution: Type-Specific Languages (TSLs)

We associate TSMs with abstract types at abstraction boundaries.

```ocaml
functor F(R : RX with syntax $rx) =
  struct
    val x : R.t = /A|T|G|C/
  end
```
functor G() => RX = struct
  type t = string
  (* ... *)
end with syntax $rx

structure G1 = G() (* G1.t has TSL $rx(G1) *)
structure G2 = G() (* G2.t has TSL $rx(G2) *)

Our Solution: Type-Specific Languages (TSLs)

We associate TSMs with abstract types at abstraction boundaries.
Conclusion

Large Languages and Syntactic DSLs considered harmful.

Typed syntax macros (TSMs) allow library providers to programmatically introduce new typed syntactic expansions in a safe, hygienic and modular manner.

Ongoing/Future Work

- Non-textual display forms in a structured editor.
- Modularly programmable type structure.