Modular Syntax

Cyrus Omar and Jonathan Aldrich
Carnegie Mellon University
{comar, aldrich}@cs.cmu.edu

1. Background

The ML module system supports **type abstraction**, which localizes reasoning about representation invariants:

Regular Expression Patterns

```
signature PATTERN = 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
```

Monadic Commands

```
signature MONAD = sig
  type 'a monad
  val ret : 'a -> 'a monad
  val bnd : 'a monad -> ('a -> 'b monad) -> 'b monad
end
```

2. The Problem

But programming against these interfaces has **high syntactic cost**:

```
(* assume P : PATTERN *)
let base = P.Or(P.Str("A"), P.Or(P.Str("T"),
  P.Or(P.Str("G"), P.Str("C"))))
let renzyme = P.Seq(P.Str("G"), P.Seq(
  P.Str("C"), P.Seq(base, P.Seq(P.Str("G"),
  P.Str("C")))))
```

```
(* assume M : MONAD,
    readInt : unit → int M.monad *)
let double_input = M.bnd (readInt ()) (fn x => M.ret
  (2*x))
```

Syntactic dialects (constructed using tools like Camlp4) can lower syntactic cost:

```
let base = /A|T|G|C/
let renzyme = /GC%{base}GC/
```

```
let double_input = do
  x ← readInt ()
  2 * x
end
```

But they are **not modular**:

Which particular module does the expansion of this derived syntax use?

If I try to combine such syntactic dialects, there might be conflicts!

3. Our Solution

Typed syntax macros (TSMs) allow library providers to programmatically introduce new syntactic expansions at a parameterized family of types:

```
syntax $pattern[Q : PATTERN] at Q.t {
  static fn(ps : ParseStream) : Exp =>
    (* … pattern parser here … *)
}
```

```
syntax $do[M : MONAD, 'a] at 'a M.monad {
  static fn(ps : ParseStream) : Exp =>
    (* … do notation parser here … *)
}
```

When applying a TSM, the characters between the delimiters are sent to the static function that the TSM defines to determine the expansion (statically!):

```
(* assume P : PATTERN *)
let base = $pattern P /A|T|G|C/
let renzyme = $pattern P /GC%{base}GC/
```

```
(* assume M : MONAD *)
let double_input = $do M int {
  x ←readInt ()
  2 * x
}
```

This solves our modularity problems!

The client specifies which module to use in the expansion as a macro parameter.

This is a language feature. Our use of delimiters ensures that there cannot be syntactic conflicts.

To further lower syntactic cost, we can associate a TSM with an abstract type directly when it becomes abstract (a **type-specific language (TSL)**):

```
structure P :> PATTERN = struct
  type t = (* … *)
  (* … *)
end with syntax $pattern
```

```
structure Option : MONAD = struct
  type 'a monad = 'a option
  fun ret x = SOME x
  fun bnd (SOME x) k = k x
      | bnd NONE k = NONE
end with syntax $do
```

When we see a delimited form not prefixed by a TSM, we use **local type inference** to apply the TSL implicitly:

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

```
let double_input : int Option.monad = {
  x ←readInt ()
  2 * x
}
```