Modular Syntax

1. Background

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

- **Regular Expression Patterns**
  
  ```ml
  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) -> 'a)
  end
  ```

- **Monadic Commands**
  
  ```ml
  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**:

- **Syntactic dialects** (constructed using tools like Camlp4) can lower syntactic cost:
  
  ```ml
  let base = PatternP /A|T|G|C/
  let renzyme = PatternP /GC%{base}GC/
  let double_input = MonadM int {
    x ← readInt ()
    2 * x
  }
  ```

- 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 TSM for regular expression patterns**
  
  ```ml
  syntax $patternQ : PATTERN at Q .t {
    static fn (ps : ParseStream) : Exp =>
    (* … pattern parser here … *)
  }
  ```

- **Syntax TSM for monadic commands**
  
  ```ml
  syntax $doM : 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!):

- **Syntactic dialects** can be used as a macro parameter.

This solves our modularity problems!

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

- **Structured definition for regular expression patterns**
  
  ```ml
  structure P :> PATTERN = struct
  type t = (* … *)
  end with syntax $pattern
  ```

- **Structured definition for monadic commands**
  
  ```ml
  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:

- **Example usage**
  
  ```ml
  let base : PatternP t = /A|T|G|C/
  let renzyme : PatternP t = /GC%{base}GC/
  let double_input : int Option.mondad = {
    x ← readInt ()
    2 * x
  }
  ```

---

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