Composable and Hygienic Typed Syntax Macros (TSMs)

Cyrus Omar
Chenglong (Stanley) Wang
Jonathan Aldrich

School of Computer Science
Carnegie Mellon University
Traditionally: Specialized syntax requires the cooperation of the language designer.
**Better approach:** an extensible language where derived syntax can be distributed in libraries.
Important Concerns

```plaintext
import kdb
import collections as c

let name = "test"
let q = {(!R)@&{&/x/:2_!x}'!R}
let z = {name => q}
```

**Hygiene**: Can I safely rename `name`?

**Composability**: Can there be parsing ambiguities (w/base language?) (w/other extensions?)

**Identifiability**: Is this controlled by a syntax extension? Which one?

**Typing Discipline**: What type do these terms have?
Type-Specific Languages (TSLs)
[Omar, Kurilova, Nistor, Chung, Potanin and Aldrich, ECOOP 2014]

```python
import kdb
import collections as c

let name = "test"
let q : kdb.Query = {(!R)@&{%2_!x}'!R}
let z : c.Map(string, kdb.Query) = {name => q}
```

**Hygiene**: Can I safely rename name?

**Identifiability**: Is this controlled by a syntax extension? Which one?

**Composability**: Can there be parsing ambiguities (w/base language?) (w/other extensions?)

**Typing Discipline**: What type do these terms have?
Limitations of TSLs

- Only one choice of syntax per type
- Cannot specify syntax for a type you don’t control
- Can’t capture idioms that aren’t restricted to one type
  - Control flow
  - API protocols
- Can’t use specialized syntax to define types themselves
Limitations of TSLs

- Only one choice of syntax per type
- Cannot specify syntax for a type you don’t control
- Can’t capture idioms that aren’t restricted to one type
  - Control flow
  - API protocols
- Can’t use specialized syntax to define types themselves
Synthetic TSMs

**Syntax**

\[
\text{Q} \Rightarrow \text{Query} = (* \ldots *)
\]

```
import kdb
import collections as c
```

**Hygiene**

Can I safely rename `name`?

```
let name = "test"
let q = kdb.Q \{\text{min} x \mod 2\text{\_til} x\}
let z : c.Map(str, kdb.Query) = \{\text{name} => q\}
```

**Identifiability**

Is this controlled by a syntax extension? Which one?

**Typing Discipline**

What type will these terms have?

**Composability**

Can there be parsing ambiguities (w/base language?) (w/other extensions?)
from web import HTML

type HTML = casetype
 TextNode of string
BodyElement of Attributes * HTML
H1Element of Attributes * HTML
(* ... *)

let greeting : HTML = H1Element({},TextNode("Hello!")))
from web import HTML

define HTML = casetype
   TextNode of string
   BodyElement of Attributes * HTML
   H1Element of Attributes * HTML
   (* ... *)

let greeting : HTML = H1Element({}, TextNode("Hello!"))
web.respond(~) (* web.respond : HTML -> () *)
<html>  (* web.respond : HTML -> () *)
    <body>
        <{greeting}>
    </body>
</html>
HTML TSL

```rust
type HTML = casetype
    TextNode of string
    BodyElement of Attributes * HTML
    H1Element of Attributes * HTML
(* ... *)

metadata = new : HasTSL
    val parser : Parser(Exp) = ~
        start <- "<body" attrs ">" start "</body>"
        fn a, c => 'BodyElement($a, $c)'
        start <- "<{" EXP "}>"
        fn spliced => spliced

let greeting : HTML = H1Element({}, TextNode("Hello!"))
web.respond(~) (* web.respond : HTML -> () *)
<html>
    <body>
        <{greeting}>
        </body>
    </html>
```

---

```rust
def import web HTML

type HTML = casetype
    TextNode of string
    BodyElement of Attributes * HTML
    H1Element of Attributes * HTML
(* ... *)

metadata = new : HasTSL
    def parser : Parser(Exp) = ~
        start <- "<body" attrs ">" start "</body>"
        def a, c => 'BodyElement($a, $c)'
        start <- "<{" EXP "}>"
        def spliced => spliced

let greeting : HTML = H1Element({}, TextNode("Hello!"))
web.respond(~) (* web.respond : HTML -> () *)
<html>
    <body>
        <{greeting}>
        </body>
    </html>
```
```javascript
from web import HTML, simpleHTML

let greeting : HTML = H1Element({}, TextNode("Hello!"))
web.respond(simpleHTML ~) (* web.respond : HTML -> () *)
>html
  >body
    < greeting
```

**HTML TSM**

```python
syntax simpleHTML => HTML = ~ (* : Parser(Exp) *)
  start <- "">body"= attrs> start>
    fn a, c => 'BodyElement($a, $c)'
  start <- ""<"= EXP>
    fn spliced => spliced
```
Limitations of TSLs

- Only one choice of syntax per type
- Cannot specify syntax for a type you don’t control
- Can’t capture idioms that aren’t restricted to one type
  - Control flow
  - API protocols
- Can’t use specialized syntax to define types themselves

Synthetic TSMs

Analytic TSMs
Analytic TSMs

```python
type bool = casetype
    True
    False

def f(error : bool, response : HTML) : HTML
    case(error)
        True => simpleHTML '<h1 Oops!'
        False => response
```
Analytic TSMs

type bool = casetype
  True
  False

syntax if = ~ (* : Parser(Exp) *)
  start <- EXP BOUNDARY EXP BOUNDARY "else" BOUNDARY EXP
  fn (e1, e2, e3) => ~
    case($e1)
    True => $e2
    False => $e3

def f(error : bool, response : HTML) : HTML
  if [error] (simpleHTML ‘Oops!’) else (response)

Can only be used in an analytic position.

4-part delimited form
Limitations of TSLs

- Only one choice of syntax per type
- Cannot specify syntax for a type you don’t control
- Can’t capture idioms that aren’t restricted to one type
  - Control flow
  - API protocols
- Can’t use specialized syntax to define types themselves

Synthetic TSMs

Analytic TSMs

Type-Level TSMs
import SQL

type StudentDB = SQL.schema ~
  *ID int
Name varchar(256)

def getByID(ID : int) : Option(Entry)
def updateByID(ID : int, entry : Entry)
def getByName(Name : string) : List(Entry)
val connection : SQL.Connection
metadata = new : HasTSL
val parser = ~
...

let db : StudentDB = ~
  url [http://localhost:2099/]
  username “test”
  password “wyvern6”
let entry = db.getByID(758)
Limitations of TSLs

- Only one choice of syntax per type
- Cannot specify syntax for a type you don’t control
- Can’t capture idioms that aren’t restricted to one type
  - Control flow
  - API protocols
- Can’t use specialized syntax to define types themselves

Synthetic TSMs

- Analytic TSMs

Type-Level TSMs

Identifiability  Composability  Hygiene

Typing Discipline / Kinding Discipline
Bidirectionally Typed Elaboration Semantics

\[ \Delta; \Gamma \vdash_\emptyset \psi \ e \rightsquigarrow i \Rightarrow \tau \]

\[ s[\text{syn}(\tau, t_{tsm})] \in \Psi \quad \text{parsestream(body)} = i_{ps} \]
\[ t_{tsm}.\text{parse}(i_{ps}) \Downarrow \text{OK}(i_{exp}) \quad i_{exp} \uparrow \hat{e} \]
\[ \Delta; \emptyset; \Gamma; \emptyset \vdash_\emptyset \hat{e} \rightsquigarrow i \Leftarrow \tau \]
\[ \Delta; \Gamma \vdash_\emptyset \text{eaptsm}[s, body] \rightsquigarrow i \Rightarrow \tau \quad (T-syn) \]

\[ s[\text{ana}(t_{tsm})] \in \Psi \quad \text{parsestream(body)} = i_{ps} \]
\[ t_{tsm}.\text{parse}(i_{ps}) \Downarrow \text{OK}(i_{exp}) \quad i_{exp} \uparrow \hat{e} \]
\[ \Delta; \emptyset; \Gamma; \emptyset \vdash_\emptyset \hat{e} \rightsquigarrow i \Leftarrow \tau \]
\[ \Delta; \Gamma \vdash_\emptyset \text{eaptsm}[s, body] \rightsquigarrow i \Leftarrow \tau \quad (T-ana) \]
Types Organize Languages

- Types represent an organizational unit for programming languages and systems.
- They can be used for more than just ensuring that programs cannot go wrong:
  - Syntax extensions (TSLs and TSMs)
  - IDE extensions (Omar et al., “Active Code Completion”, ICSE 2012)
  - Type system extensions (talk to me)

II. Survey

To validate our general conceptualization of active code completion, develop concrete criteria to constrain our system and palette designs, and create a list of use cases to justify this effort, we began by conducting a large survey of professional software developers.

A. Participants

We recruited participants for this survey primarily from a popular programming-related discussion forum hosted on the popular website reddit.com [12]. An additional 22 participants were computer science graduate students at CMU. Recruitment materials in both cases stated that we were seeking developers “familiar with an object-oriented programming language like Java, C# or Visual Basic and an integrated development environment like Eclipse or Visual Studio”. Participants were told that the survey would take approximately 20 minutes to complete, and no reward was offered. Of the 696 people who started the survey, 473 participants (68%) completed it. We examine the responses from completed surveys only in the analyses below.

B. Familiarity with Programming Languages and Editors

We first asked participants about their level of familiarity with several programming languages, on a five-point Likert scale. 61.1% of the participants indicated that they were an expert in at least one language, and an additional 35.7% were “very familiar” with at least one language. On average, participants rated themselves as very familiar with Java, C, C++ and JavaScript, familiar with C#, Python and PHP and somewhat familiar with Visual Basic and Perl.

We also asked participants to select which integrated development environments (IDEs) and code editors that they were familiar with. The Eclipse IDE was familiar to 87.1% of participants. This was followed by Visual Studio at 66.0%, Vi/Vim at 53.7%, Netbeans at 37.7%, Emacs at 24.8% and IntelliJ IDEA at 16.4%. Participants could also enter “other” choices and a number of editors and IDEs were entered, including Xcode, Textmate and Notepad++.