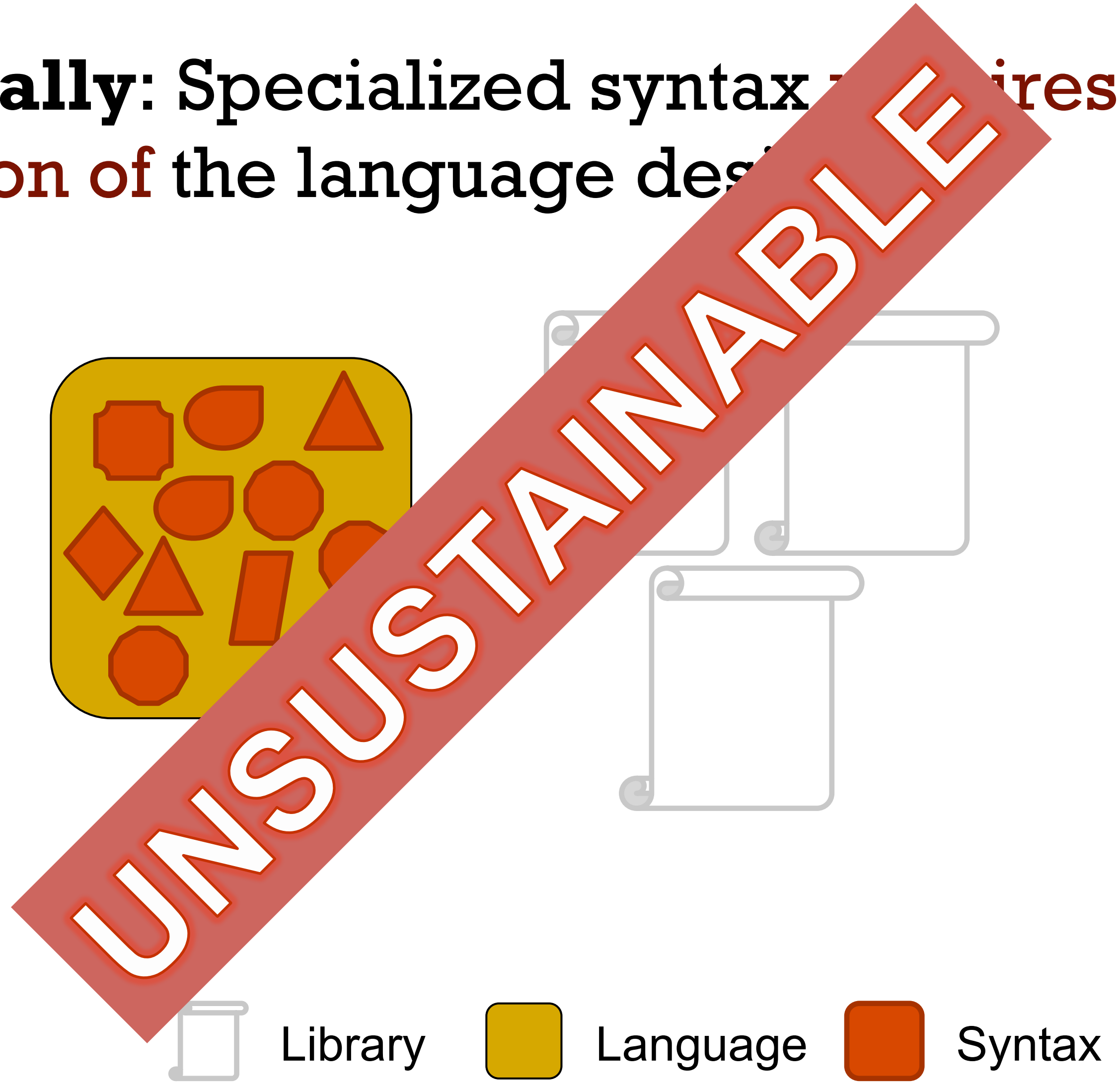
[DSLDI14]

Composable and Hygienic <u>Typed Syntax Macros</u> (TSMs)

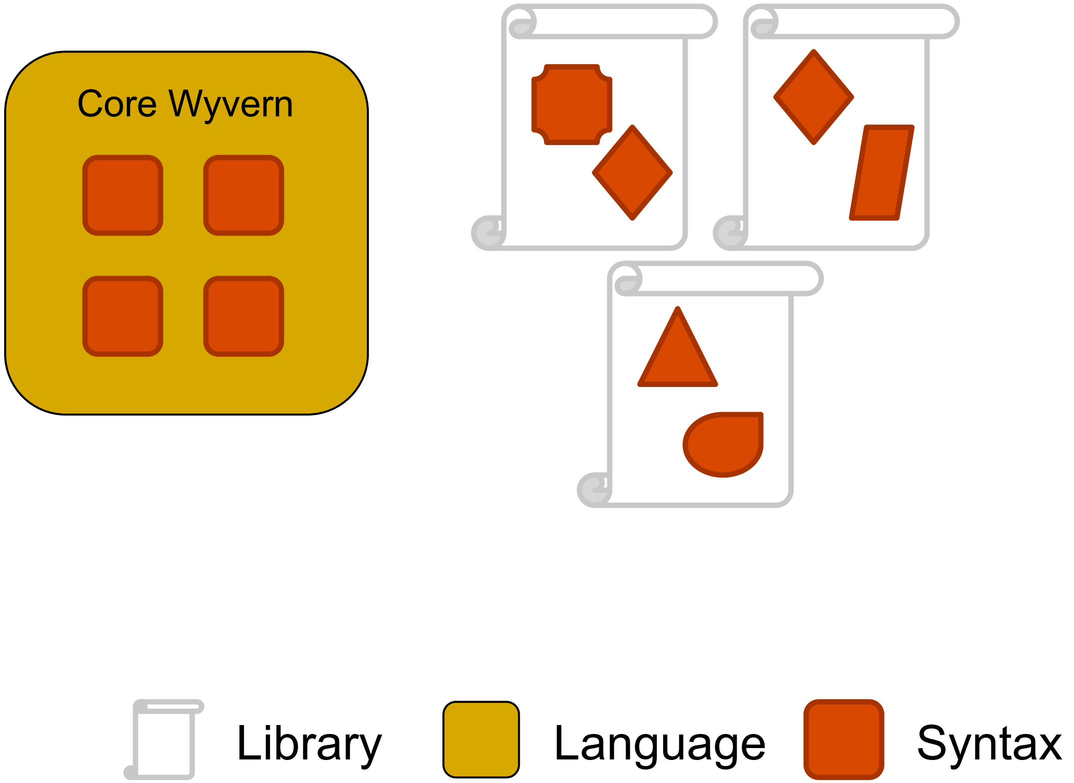
Cyrus Omar
Chenglong (Stanley) Wang
Jonathan Aldrich

School of Computer Science
Carnegie Mellon University

Traditionally: Specialized syntax r… ires the cooperation of the language des…



Better approach: an extensible language where derived syntax can be distributed in libraries.



Important Concerns

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

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

Hygiene: Can I safely rename `name`?

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

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

Typing Discipline: What type do these terms have?

Type-Specific Languages (TSLs)

[Omar, Kurilova, Nistor, Chung, Potanin and Aldrich, **ECOOP 2014**]

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

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

Hygiene: Can I safely rename `name`?

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

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

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

Synthetic TSMs

- Can't capture idioms that aren't restricted to one type
  - Control flow
  - API protocols

Analytic TSMs

- Can't use specialized syntax to define types themselves

Type-Level TSMs

Synthetic TSMs

```
syntax Q => Query = (* ... *)
```

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

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

Hygiene: Can I safely rename `name`?

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

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

Typing Discipline: What type will these terms have?

HTML

```
from web import HTML
```

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

```
let greeting : HTML = H1Element({}, TextNode(“Hello!”))
```

HTML TSL

```
from web import HTML
```

```
type 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>
    <body>
      <{greeting}>
    </body>
  </html>
```

HTML TSL

```
from web import HTML
```

```
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>
```

HTML TSM

```
from web import HTML, simpleHTML
```

```
syntax simpleHTML => HTML = ~ (* : Parser(Exp) *)
  start <- “>body”= attrs> start>
    fn a, c => ‘BodyElement($a, $c)’
  start <- “<“= EXP>
    fn spliced => spliced
```

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

Limitations of TSLs

- ✅ Only one choice of syntax per type
- ✅ Cannot specify syntax for a type you don't control

Synthetic TSMs

- Can't capture idioms that aren't restricted to one type
  - Control flow
  - API protocols

Analytic TSMs

- Can't use specialized syntax to define types themselves

Analytic TSMs

```
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
```

Can only be used in an **analytic** position.

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

4-part delimited form

Limitations of TSLs

- ✅ Only one choice of syntax per type
- ✅ Cannot specify syntax for a type you don't control

Synthetic TSMs

- ✅ Can't capture idioms that aren't restricted to one type
  - Control flow
  - API protocols

Analytic TSMs

- Can't use specialized syntax to define types themselves

Type-Level TSMs

Type-Level TSMs

```
syntax schema :: * with metadata : HasTSL = ~ (* Parser(Type * HasTSL) *)
```

Kinding Discipline: What kind will these types have?

```
import S

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

```
  type Entry = objtype
    val ID : int
    val Name : string
  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

Synthetic TSMs

- Can't capture idioms that aren't restricted to one type
  - Control flow
  - API protocols

Analytic TSMs

- Can't use specialized syntax to define types themselves

Type-Level TSMs

Identifiability **Composability** **Hygiene**

Typing Discipline / Kinding Discipline

Bidirectionally Typed Elaboration Semantics

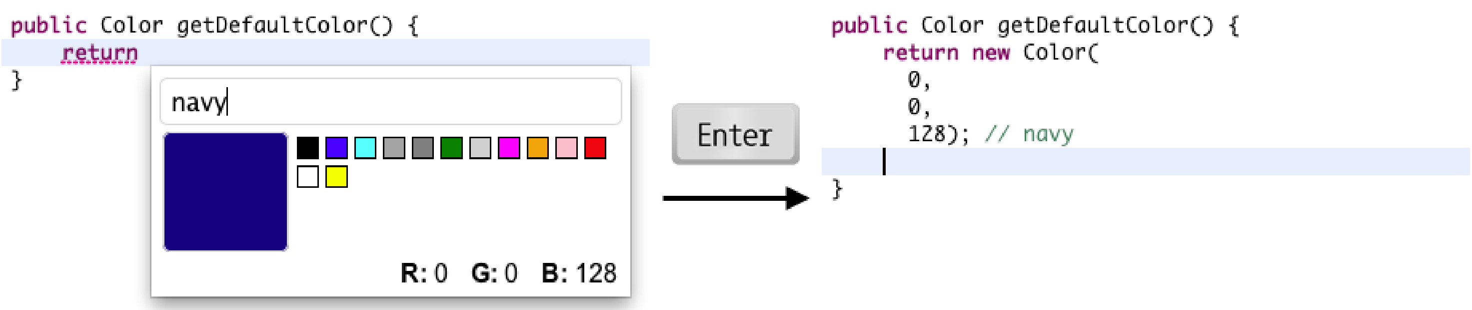
$$\boxed{\Delta; \Gamma \vdash_{\Theta}^{\Psi} e \leadsto i \Rightarrow \tau}$$

$$\frac{\begin{array}{c} s[\mathbf{syn}(\tau, i_{tsm})] \in \Psi \qquad \mathsf{parsestream}(body) = i_{ps} \\ i_{tsm}.parse(i_{ps}) \Downarrow OK(i_{exp}) \qquad i_{exp} \uparrow \hat{e} \\ \Delta; \emptyset; \Gamma; \emptyset \vdash_{\Theta}^{\Psi} \hat{e} \leadsto i \Leftarrow \tau \end{array}}{\Delta; \Gamma \vdash_{\Theta}^{\Psi} \mathbf{eaptsm}[s, body] \leadsto i \Rightarrow \tau} \text{(T-syn)}$$

$$\frac{\begin{array}{c} s[\mathbf{ana}(i_{tsm})] \in \Psi \qquad \mathsf{parsestream}(body) = i_{ps} \\ i_{tsm}.parse(i_{ps}) \Downarrow OK(i_{exp}) \qquad i_{exp} \uparrow \hat{e} \\ \Delta; \emptyset; \Gamma; \emptyset \vdash_{\Theta}^{\Psi} \hat{e} \leadsto i \Leftarrow \tau \end{array}}{\Delta; \Gamma \vdash_{\Theta}^{\Psi} \mathbf{eaptsm}[s, body] \leadsto i \Leftarrow \tau} \text{(T-ana)}$$

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)



(a) (b)

  - Type system extensions (talk to me)