Composable and Hygienic Typed Syntax Macros[*]

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

ABSTRACT

Syntax extension mechanisms are powerful, but reasoning about syntax extensions can be difficult. Recent work on *type-specific languages (TSLs)* addressed reasoning about composition, hygiene and typing for extensions introducing new literal forms. We supplement TSLs with *typed syntax macros (TSMs)*, which, unlike TSLs, are explicitly invoked to give meaning to delimited segments of arbitrary syntax. To maintain a typing discipline, we describe two flavors of term-level TSMs: synthetic TSMs specify the type of term that they generate, while analytic TSMs can generate terms of arbitrary type, but can only be used in positions where the type is otherwise known. At the level of types, we describe a third flavor of TSM that generates a type of a specified kind along with its TSL and show interesting use cases where the two mechanisms operate in concert.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Extensible languages*

Keywords

extensible syntax; macros; hygiene; type inference

1. INTRODUCTION

One way programming languages evolve is by introducing *syntactic sugar* that captures common idioms more concisely and naturally. In most contemporary languages, this is the responsibility of the language designer. Unfortunately, the designers of general-purpose languages do not have strong incentives to capture idioms that arise only situationally, motivating research into mechanisms that allow the users of a language to extend it with new syntactic sugar themselves.

Designing a useful syntax extension mechanism is non-trivial because the designer can no longer comprehensively check that parsing ambiguities cannot arise and that desugarings are semantically well-behaved. Instead, the extension mechanism must provide several key guarantees:

[*]This paper uses color for clarity of exposition.

SAC '15 April 13 - 17 2015, Salamanca, Spain
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3196-8/15/04 ...$15.00.
http://dx.doi.org/10.1145/2695664.2695936

Composability The mechanism cannot simply allow the base language's syntax to be modified arbitrarily due to the potential for parsing ambiguities, both due to conflicts with the base language and, critically, between extensions (e.g. extensions adding support for XML and HTML).

Hygiene The desugaring logic associated with new forms must be constrained to ensure that the meaning of a valid program cannot change simply because some of the variables have been uniformly renamed (manually, or by a refactoring tool). It should also be straightforward to identify the binding site of a variable, even with intervening uses of sugar. These two situations correspond to inadvertent variable capture and shadowing by the desugaring.

Typing Discipline In a rich statically typed language, which will be our focus in this work, determining the type a sugared term will have, and analagously the *kind* a type will have (discussed further below), should be possible without requiring that the desugaring be performed, to aid both the programmer and tools like type-aware code editors.

Most prior approaches to syntax extension, discussed in Sec. 5, fail to simultaneously provide all of these guarantees. Recent work on *type-specific languages (TSLs)* makes these guarantees, but in a limited setting: library providers can define new literal syntax by associating parsing and desugaring logic with type declarations [13]. Local type inference, specified as a bidirectional type system [15], controls which such TSL is used to parse the bodies of literal forms. The available delimiters are fixed by the language, but the bodies of literal forms can be arbitrary, so TSLs are flexible, and this approach guarantees composability and maintains the typing discipline by construction. The semantics given also guarantees hygiene. We will review in Sec. 2.

While many forms of syntactic sugar can be realized as TSLs, there remain situations where TSLs do not suffice:

(i) Only a single TSL can be associated with a type, and only when it is declared, so alternate syntactic choices (which are common [17]), or syntax for a type that is not under a programmer's control, cannot be defined.
(ii) Syntax cannot be associated with types that are not identified nominally (e.g. arrow types).
(iii) Idioms other than those that arise when introducing a value of a type (e.g. those related to control flow or API protocols) cannot be captured
(iv) Types cannot themselves be declared using specialized syntax.

Contributions In this paper, we introduce *typed syntax macros (TSMs)*, which supplement TSLs to handle these scenarios while maintaining the crucial guarantees above.

```
1  casetype HTML
2    Empty
3    Seq of HTML * HTML
4    Text of String
5    BodyElement of Attributes * HTML
6    H1Element of Attributes * HTML
7    StyleElement  of Attributes * CSS
8    (* ... *)
9    syntax = ~ (* : Parser(Exp) *)
10     start <- '<body' attributes '>' start '</body>'
11       fn atts, child => 'BodyElement(($atts, $child))'
12     start <- '<(' EXP ')>'
13       fn e => e
14     (* ... *)
15
16 let heading : HTML = H1Element({}, Text("My Heading"))
17 serve(~) (* serve : HTML -> Unit *)
18   <body id="doc1">
19     <(heading)>
20     <p>My first paragraph.</p>
21   </body>
```

Figure 1: A case type with an associated TSL.

Our specific contributions are as follows:

1. We introduce TSMs first at the term level in Sec. 3. To maintain a typing discipline, there are two flavors of term-level TSMs: *synthetic TSMs* can be used anywhere, while *analytic TSMs* can only be used where the expected type of the term is otherwise known.
2. We next turn to type declarations in Sec. 4. Type-level TSMs generate both a type of a specified *kind*, maintaining the *kinding discipline* that governs type parameter application, and also the TSL associated with it, so TSLs and TSMs can operate in concert.
3. Both TSLs and TSMs leverage lightweight delimiters to separate syntax extensions from the host language. We supplement the delimited forms previously defined to support additional idioms.

We more specifically compare our work to related work in Sec. 5 and conclude in Sec. 6. A more detailed type-theoretic treatment of these mechanisms is available in an accompanying technical report [14].

2. BACKGROUND

2.1 Wyvern

We will present TSMs in the context of the simple variant of the Wyvern programming language introduced previously to describe TSLs [13], making only minor changes that we will note as they come up. Wyvern is a statically typed language with features from both the functional and object-oriented traditions and has a layout-sensitive syntax.

An example of a type encoding the tree structure of HTML is declared in Figure 1. The type `HTML` is a *case type*, with cases for each HTML tag and additional cases for an empty document, a sequence of nodes and a text node. Case types are similar to datatypes in an ML-like language (in type-theoretic terms, recursive labeled sum types). We introduce a value of type `HTML` on line 17 by naming a case and providing an argument of the type the case declares.

Wyvern also supports declaring object types, e.g. `Parser` in Figure 2 (discussed below). Object types declare fields and methods via `val` and `def`, respectively. Values of object type are introduced with **`new`**, which is a *syntactic forward reference*: it can appear once per line, at the term position where an object is needed. The next indented block gives the field values and method implementations (Figure 7 shows some examples). Objects are similar to functional records.

```
objtype Parser(T)
  def parse(ParseStream) : Result(T)
  syntax = (* ... parser generator syntax ... *)

casetype Result(T)
  OK of T
  Error of String * Location

casetype Exp
  Var of ID
  Fn of ID * Exp
  Ap of Exp * Exp
  Ascription of Exp * Type
  CaseIntro of ID * Exp
  (* ... *)
  Spliced of ParseStream (* see Sec. 3.3 *)
  syntax = (* ... exp quasiquotes ... *)

casetype Type
  Declared of ID
  Objtype of List(MemberDecl)
  Casetype of List(CaseDecl)
  Arrow of Type * Type
  (* ... *)
  TyVar of ID (* see Sec. 4.1 *)
  TyFn of ID * Type
  TyAp of Type * Type
  Spliced of ParseStream (* see Sec. 4.2 *)
  syntax = (* ... type quasiquotes ... *)
```

Figure 2: A portion of the Wyvern prelude relevant to TSLs and TSMs.

We assume standard types like `String`, `List` and `Option` are ambiently available in a *prelude*. Types declarations are *generative*, i.e. declared types are identified nominally (like datatypes in ML, or classes in Java). Declarations can also include type parameters, e.g. `List` would declare one type parameter. To be more precise, we say that `List` is a *type constructor*, in that applying `List` to a type produces a particular list type, e.g. `List(String)`. Types have *kind* `Ty`, while type constructors have arrow kind, e.g. `List` has kind `Ty -> Ty`. For concision, we use the phrase *declared type* for names having any kind.

Some types are identified structurally, e.g. tuple types like `HTML * HTML` and function types like `HTML -> Unit`. When object and case types are written anonymously, rather than via a declaration, they are also identified structurally.

2.2 Type-Specific Languages (TSLs)

Introducing a value of type `HTML` using general-purpose syntax like that shown on line 17 of Figure 1 can be tedious. Moreover, there is standard concrete syntax for HTML that might be preferable for reasons of familiarity or backwards compatibility. To allow for this, we associate a *type-specific language* with the `HTML` type. We see this TSL being used on lines 18-22 of Figure 1. On line 18, we call the function `serve`, which we assume has type `HTML -> Unit`. Rather than explicitly constructing a term of type `HTML` as the argument, we use the *forward referenced literal form* ~. The *body* of the literal consists of the text in the indented block beginning on the next line, stripped of the leading indentation. In effect, whitespace is serving as a delimiter.

We could equivalently have used other *inline delimiters*, e.g. curly braces or single quotes, though we would then need to follow the restrictions described in Figure 3. For example, we could have written line 17 equivalently as:

```
    val heading : HTML = '<h1>My Heading</h1>'
```

```
'body here, ''inner single quotes'' must be doubled'
[body here, [inner braces] must be balanced]
{body here, {inner curly braces} must be balanced}
~ (* can appear at any expression position *)
  forward referenced body here, leading indent stripped
(parentheses_delimitv_spliced_terms_in_TSM_arguments(~))
  so forward references can propagate out
[adjacent] {delimited forms} or [those] separated ~ form
  by identifiers create a single multipart delimited
```

Figure 3: Available delimited forms in Wyvern. The parentheses and multipart delimited forms are novel, discussed in Sec. 3.

The first phase of parsing leaves the bodies of such delimited forms unparsed. They are parsed and elaborated during typechecking. More specifically, when the semantics encounters any such literal form, the syntax associated with the declared type that the literal is being analyzed against, here `HTML`, is used to desugar it before continuing.

Such syntax is associated with a declared type as seen on lines 9-14 of Figure 1 by writing `syntax = e`, where `e` is a parser of type `Parser(Exp)`.[1,2] Per Figure 2, a parser defines a `parse` function that transforms a `ParseStream` based on the body to a `Result(Exp)`, which is either a desugaring, encoded as a value of type `Exp`, or an indication of a parse error. Rather than writing a `parse` function explicitly, we make use of the fact that `Parser` itself has a TSL associated with it providing a static *grammar-based parser generator*. This allows us to create a `Parser(Exp)` by specifying a number of productions each of which is followed by a Wyvern function taking in the elaborations of each constituent non-terminal and producing the final elaboration of type `Exp`. The non-terminal `start` serves as the starting non-terminal.

The types `Exp` and `Type` encode the abstract syntax of Wyvern terms and types, respectively. To make code generation more straightforward, these types are equipped with TSLs that provide *quasiquotation* [16, 3, 4]: terms of these types can be written using Wyvern's usual concrete syntax, extended with unquote forms `$x` and `$(e)`, which splice in variable `x` and term `e`, respectively.

Splicing can be supported by any TSL. For example, the TSL for `HTML` uses the delimiters `<(` and `)>` to mean "splice in the enclosed term of type `HTML` here", as seen on line 19 of Figure 1. This is supported because a parser can request that some portion of the parse stream be treated as a host language term, type or variable. The cases `Spliced` in `Exp` and `Type` track these spliced portions of the parse stream. This is the basis of the hygiene mechanism in [13]. The parser generator provides the non-terminals `EXP`, `ID` and `TYPE`, which generate these spliced forms internally. For example, `EXP` is used on line 13 of Figure 1 to implement HTML splicing.

For expository purposes, we color the bodies of delimited forms a color corresponding to the TSL or TSM being used, identified when declared. Base language terms, including spliced base terms, are colored black.

3. TERM-LEVEL TSMs

Having introduced the necessary background, we will now describe term-level typed syntax macros and illustrate their use in situations where TSLs are not suitable.

[1]In [13], we gave a more general *metadata*-based mechanism, but we give a simpler TSL-specific mechanism here.

[2]For expository purposes, we include type ascriptions that are not strictly needed in comments throughout the paper.

```
1  syntax simpleHTML for HTML = ~ (* : Parser(Exp) *)
2    start <- '>body'= attributes> start>
3      fn atts, child => 'BodyElement(($atts, $child))'
4    start <- '<'= EXP>
5      fn e => e
6    (* ... *)
7  let heading = simpleHTML '>h1 My Heading'
8  serve(simpleHTML ~)
9    >body[id="doc1"]
10     < heading
11     >p My first paragraph
```

Figure 4: A synthetic TSM providing alternative syntax for the `HTML` type in Figure 1.

3.1 Synthetic TSMs

TSMs are defined using the **`syntax`** keyword. Figure 4 shows a synthetic TSM, `simpleHTML`, being defined and used. This TSM defines an alternative layout-sensitive syntax for HTML. The clause **`for`** `HTML` indicates that valid invocations of the TSM will necessarily elaborate to a term of type `HTML`.

Like defining a TSL, defining a TSM requires defining a parser, i.e. a statically evaluated value of type `Parser(Exp)`. We again do so using the parser generator associated with `Parser`. In this case, we take advantage of its support for *Adams grammars*, which allow declarative specifications of layout-sensitive grammars using *layout constraints* within productions [2]. Here, the suffix `=` indicates that the leftmost column (on any line) occupied by the annotated terminal or non-terminal must occur at the same column as the parent and `>` indicates that it must be further indented.

Whereas TSLs are invoked implicitly based on local type inference, invoking a TSM is similar to function application: the name of the TSL is followed by a delimited form from Figure 3. The body of the delimited form is parsed and elaborated by the parser the TSM defines. For example, we use single quotes on line 7 of Figure 4. Notice that we did not need a type annotation on `heading`. On lines 8-11, we again invoke `simpleHTML`, this time using the same forward referenced delimited form described above. Lines 16-21 of Figure 1 are semantically equivalent to lines 7-11 of Figure 4, differing only syntactically.

Synthetic TSMs address issue (i) from Sec. 1: modularly and composably defining more than one choice of syntax for a type that either has a TSL already, or a type which a user cannot modify. Compared to the alternative solution of defining a type `HTML2` with a TSL that has the desired syntax, and a function mapping from `HTML2` to `HTML`, TSMs avoid declaration duplication and the $\mathcal{O}(n)$ runtime overhead.

Synthetic TSMs also address issue (ii) because they can define syntax for types identified structurally (e.g. functions, pairs and anonymous objects), not just nominally.

3.2 Analytic TSMs

TSLs and synthetic TSMs are not suitable for expressing idioms that are valid at many types, i.e. issue (iii) from Sec. 1. As perhaps the simplest example, consider the case type encoding booleans declared in Figure 5. Explicit case analysis on booleans is considered unnecessarily verbose, so many languages defining booleans in this manner build in an `if` construct as a desugaring. Rather than having to build this in to Wyvern, however, we can express it as an analytic TSM, on lines 4-9. These are distinguished from synthetic TSMs in that they don't declare a type, here because the type of an `if` expression is determined by its branches.

```
 1 casetype Bool
 2   True
 3   False
 4 syntax if = ~ (* : Parser(Exp) *)
 5   EXP BOUNDARY EXP BOUNDARY 'else' BOUNDARY EXP
 6     fn guard, branch1, branch2 => ~ (* : Exp *)
 7       case $guard
 8         True => $branch1
 9         False => $branch2
10 def testIf(ok : Bool) : HTML
11   if (ok) (simpleHTML ~) else ('<h1>Not OK!</h1>')
12     >h1 Everything is OK!
```

Figure 5: An analytic TSM providing a conventional syntax for `if` based on case analysis. Lines 11-12 demonstrate multipart delimited forms.

We see `if` being used on lines 10-12 of Figure 5 with a *multipart delimited form*. Each *part* can be either a single delimited form (e.g. the guard and the two branches) or one or more intervening identifiers (e.g. `else`). The body is generated by concatenating the bodies of the parts and inserting between them a special boundary character outside the normal character set. We call this character `BOUNDARY` in our parser generator (line 5). Intervening identifiers can be thought of as having implicit delimiters around them, e.g. `if [e1] else [e2]` and `if [e1] [else] [e2]` express the same body, `e1·else·e2`, where · is the boundary character.

For the branches in our example, we used parentheses-delimited parts. In a TSM application, this means that the part consists of a single spliced term and no other syntax. Because the parser can assume this, forward references can be identified prior to typechecking and thus be allowed to escape, as we see in the "then" branch in our example: the body is on the next line. Had we used, for example, square brackets, then we would need to write the example like this:

```
def testIf2(ok : Bool) : HTML
  if (ok) [simpleHTML ~
    >h1 Everything is OK!
  ] else ['<h1>Not OK!</h1>']
```

An analytic TSM can only be used in a position where the type is otherwise known, e.g. on line 10 due to the return type annotation. This is to maintain the typing discipline: we do not need to expand the TSM to know what type it will have, just as with synthetic TSMs and TSLs.

Although we believe this trade-off is worthwhile, another point in the design space is to permit a special signifier that can be used to explicitly allow analytic TSMs to be used in synthetic positions. For example, we might permit a postfix asterisk, `if*`. The type the term will have then requires a deeper understanding of the TSM in question (e.g. by knowing how it elaborates, or based on a "derived" typing rule that the providers of `if` assert or prove [11]).

3.3 Hygiene

The hygiene mechanism for term-level TSMs is identical to that for TSLs. The details are in [13]. To summarize: spliced sub-terms are marked as such in the generated code, and only these can refer to variables in the surrounding scope. No new variables can be introduced into their scope.

4. TYPE-LEVEL TSMs

We now turn our attention to TSMs at the level of type declarations. Our example will be a simple object-relational mapping (ORM) syntax, shown being used in Figure 6. An

```
decltype EmployeesDB = schema ~
  *ID int
  Name varchar

let db : EmployeesDB = ~
  connect to ~
    mysql://localhost:3306
  table    "Employees"
  username "user1"
  password "001"
db.getByID(758) (* : Option(EmployeeDB.Entry) *)
```

Figure 6: The usage of a type-level TSM and the TSL it generates to enable a simple ORM.

ORM provides an object-oriented interface to a relational database, generated from a database *schema*. For example, the schema in Figure 6 specifies a table with two columns, `ID` and `Name`, holding values of the SQL data types `int` and `varchar`. The `ID` column is marked with an asterisk as being a *primary key*, meaning that it must be unique across rows.

ORMs typically rely on an external code generator, which can hinder code comprehension because the fully elaborated interface is exposed directly to the programmer, obscuring the simpler schema by moving it to an external resource. By instead using the type-level TSM `schema`, the interface shown in Figure 7 is generated from the schema during compilation, using a language-integrated mechanism. More specifically, in Figure 7 the type member `Entry` declares a field for each column in the schema, with its type generated based on a mapping from SQL types to Wyvern types (not shown). Moreover, for each column `C`, a method named `getByC` is also generated. The return type of this method is an option type if the column is a primary key (reflecting the uniqueness invariant) or a list otherwise. There are also fields for connection parameters.

As discussed in Section 2, declared types in Wyvern can define a TSL. A type-level TSM generates not only a type, but also its TSL. Here, the TSL that `schema` generates is shown being used on lines 5-10 of Figure 8 to create a value of the generated type `EmployeesDB`. Only the per-database parameters need to be provided.

The definition of `schema` is shown in Figure 8. Type-level TSMs must specify the *kind* of type-level term that will be generated, here $*$ because the type takes no parameters. This is to maintain a *kinding discipline*: knowing how many type parameters a type takes does not require desugaring its declaration. Elaboration is then defined by a parser of type `Parser(Type*Parser(Exp))`, i.e. it must map the body to a reified type-level term and the TSL parser.

Here, we generate the type using type quasiquotation by mapping over the column specifications parsed out of the body on lines 37-43 (using the same color for both `Type` and `MemberDecl` for simplicity). We assume a mapping from SQL types to Wyvern types, `ty_from_sqlty`, not shown. Starting on line 20, we then generate the TSL as described in Sec. 2. Note that the implementations of the methods generated on lines 4-19 are filled in by the TSL generated on lines 20-35.

4.1 Recursive and Parameterized Types

In this example, there were no type parameters and the generated type was not recursive. To understand how we can generate such types, we must first break down our type declaration mechanism in a bit more detail. Recall that type declarations using **`casetype`** or **`objtype`** were generative. In

```
decltype EmployeesDB = objtype
  decltype Entry = objtype
    val ID : Int
    val Name : String
  val connection : URL
  val username : String
  val password : String
  (* ... *)
  def getByID(Int) : Option(Entry)
  def getByName(String) : List(Entry)
  syntax = (* ... generated TSL, cf Figure 8 ... *)

let db : EmployeesDB = new
  val connection = new
    val domain = "localhost"
    (* ... *)
  val username = "user1"
  val password = "001"
  (* ... *)
  def getByID(x)
    (* send appropriate query *)
db.getByID(758)
```

Figure 7: The elaboration of Figure 6.

```
 1 syntax schema :: Ty = ~ (*:Parser(Type*Parser(Exp))*)
 2   start <- columns
 3     fn cols (* : List(Bool*Label*Type) *) =>
 4       let ty : Type = ~
 5         objtype
 6           type Entry = objtype
 7             $(
 8               map(cols, fn (primary, lbl, ty) => ~)
 9                 val $lbl :  $ty
10             )
11           val connection :  URL
12           val username :  String
13           val password :  String
14           (* ...  *)
15           $(
16             map(cols, fn (primary, lbl, ty) => ~)
17               def getBy$lbl($ty) :  $(if (primary) (
18                 'Option(Entry)') else ('List(Entry)'))
19           )
20       let tsl : Parser(Exp) = ~
21         start <- ("connect to"= EXP>
22                  "table"= EXP>
23                  "username"= EXP>
24                  "password"= EXP>)
25           fn url, un, pw, table => ~
26             new
27               val connection = $url
28               val username = $un
29               val password = $pw
30               (* ...  *)
31               $(
32                 map(cols, fn (primary, lbl, ty) => ~)
33                   def getBy$lbl(x)
34                     (* send appropriate query *)
35               )
36       (ty, tsl)
37   columns <- column
38     fn column => Cons(column, Nil)
39   columns <- column= columns=
40     fn column, columns => Cons(column, columns)
41   column <- "*"?  ID ID
42     fn primary, lbl, sqlty =>
43       (primary, lbl, ty_from_sqlty(sqlty))
```

Figure 8: The definition of a type-level TSM.

fact, these declaration forms are (built in) syntactic sugar for the general generative type declaration form, **`decltype`**. The right-hand side of this declaration form consists of a type level function that takes in a self-reference followed by the type parameters before producing an anonymous type on the right. For example, `List` is actually declared as follows:

```
decltype List = tyfn S::Ty->Ty => tyfn T::Ty => casetype
  Nil
  Cons of T * S(T)
```

Note that the type-level function after the equals sign as a whole has kind `(Ty -> Ty)-> (Ty -> Ty)`. More generally, for a type with n type parameters, the type-level function must have kind `(Ty ->`$_n$ `Ty)-> (Ty ->`$_n$ `Ty)`, where the subscript n indicates n arguments of kind `Ty`. A type-level TSM must generate such a function. On line 5 of Figure 8, the mechanism allowed that the self-reference be omitted because the type was not recursive, but we could have written **`tyfn`** `S :: Ty =>` **`objtype`**. Note that the name of the type is determined by the client of the TSM. The type variable `S` here is relevant only inside the TSM definition. We discuss mutually recursive types in the technical report [14].

4.2 Hygiene

The previous work on TSLs did not handle parameterized types or parametric polymorphism, so there was no notion of a type variable there. With these features included, we must also have a notion of hygiene with respect to type variables. We can take an analagous approach to the one taken for term variables, marking spliced sub-terms and giving only these access to the surrounding type variable context when kind checking the generated type. The details are in the technical report [14].

5. RELATED WORK

Unlike other work on library-integrated syntax extension mechanisms, e.g. SugarJ [6] and its subsequent variations [7], protean operators [9], mechanisms like those found in Coq [12, 8] and Nemerle [19] and various language-external mechanisms like Camlp4 [10], we do not permit the syntax of the base language to be extended directly. Instead, we build on the delimited forms used for type-specific languages [13]. Using delimiters to separate extensions from the base language guarantees that any combination of libraries can be imported and used together (i.e. *composed*), without "link-time" parsing ambiguities, because different extensions can only interact via the host language using splicing.

Schwerdfeger and Van Wyk have shown a composable analysis for syntax specified using an LR parser generator with a context-aware scanner [18]. Like our work, they rely on a unique starting token to identify a language, but perform a sophisticated analysis on follow sets of non-terminals to guarantee composability. Our use of fixed delimiters is simpler – no analysis need to be run at all – and allows for arbitrary parse functions. A parser generator (in our case, based on Adams grammars [2]) is simply a TSL atop this mechanism. Using a synthetic TSM, different parser generator formalisms could be defined (e.g. for regular languages, a simpler mechanism using regular expressions might suffice).

Macro systems have a long history in the LISP family of languages. These typically only permit rewriting existing syntax (typically, a variant on S-expressions), rather than introducing arbitrary new syntax, though *reader macros* do allow some syntax extensions as well, albeit without strict composability guarantees [20]. We use the phrase *syntax macro* to describe our work because like macros, TSMs are invoked explicitly by name and are used to generate code, but this occurs during typechecking. The initial parsing phase separates delimited forms but leaves bodies unparsed.

Most existing syntax extension mechanisms don't support a typing discipline. A notable exception is work by Lorenzen and Erdweg, who described a mechanism for automatically proving the admissibility of derived typing rules for syntax extensions [11]. Integrating such facilities into TSMs would be an interesting avenue for future work.

Macro systems that do consider the typing discipline, e.g. in Scala [4], do exist but as just mentioned, do not support syntax extension. In Scala, *black box macros* are similar to synthetic TSMs in that they specify a type signature. Analytic TSMs can be seen as a restriction on the use of *white box macros* (disabled by default in recent versions of Scala) to analytic positions.

Concerns about hygiene have been well-studied in the macros communities, e.g. in Scheme [5]. Because we defer parsing of delimited forms to typechecking time, our formalization of the hygiene mechanism can be cleanly specified in terms of access to typing contexts and, uniquely, we track portions of the parse stream that correspond to spliced terms implicitly. We also considered hygiene at the type level here.

Standard ML of New Jersey supports quotation and antiquotation using the concept of *fragment lists* [1]. This is composable and obeys a typing discipline, but has three main problems:

1. Parsing of fragment lists must still occur at run-time. This is perhaps the biggest difference relative to our mechanism, which introduces new static desugarings.
2. Syntax that uses backticks or carets is difficult to introduce in this way. In our mechanism, a variety of delimiters, notably including layout, can be used, and the extension itself determines how antiquotation (i.e. splicing) is initiated, so such difficulties can be avoided.
3. Only one type of subterm can occur inside antiquotes. One must define and use a datatype if there may be different types of subterms (e.g. our HTML example above). This can again defeat part of the purpose of introducing concrete syntax.

6. CONCLUSION AND FUTURE WORK

Taken together, TSLs and TSMs represent what we see as a new "high water mark" in expressive power, especially within the space of systems that guarantee *composability*, *hygiene* and *typing discipline* described in Sec. 1 and are rigorously specified in type theoretic terms, as we show in the accompanying technical report [14].

There remain several promising avenues for future work. While synthetic TSMs as shown allow specifying syntax for any particular type, we did not show any way to specify syntax for all types having a common type constructor (e.g. syntax for all lists). Similarly, we do not show how to specify syntax for abstract types. Another promising avenue for future work is to strengthen splicing so that a single delimiter can support multiple types of spliced terms (e.g. allowing both strings and HTML to be spliced into HTML). Enabling editor support is also an important direction for future work.

ACKNOWLEDGEMENTS

We thank the anonymous referees for helpful suggestions, and acknowledge the support of the National Security Agency lablet contract #H98230-14-C-0140 and National Science Foundation grant #CCF-1116907.

7. REFERENCES

[1] SML/NJ Quote/Antiquote. `http://www.smlnj.org/doc/quote.html`.

[2] M. D. Adams. Principled Parsing for Indentation-Sensitive Languages: Revisiting Landin's Offside Rule. In *POPL*, 2013.

[3] A. Bawden. Quasiquotation in Lisp. In *Partial Evaluation and Semantic-Based Program Manipulation*, 1999.

[4] E. Burmako. Scala Macros: Let Our Powers Combine!: On How Rich Syntax and Static Types Work with Metaprogramming. In *Proceedings of the 4th Workshop on Scala*, SCALA, 2013.

[5] R. K. Dybvig, R. Hieb, and C. Bruggeman. Syntactic abstraction in Scheme. *Lisp Symb. Comput.*, 5(4):295–326, Dec. 1992.

[6] S. Erdweg, T. Rendel, C. Kastner, and K. Ostermann. SugarJ: Library-based syntactic language extensibility. In *OOPSLA*, 2011.

[7] S. Erdweg and F. Rieger. A framework for extensible languages. In *GPCE*, 2013.

[8] T. Griffin. Notational definition-a formal account. In *Logic in Computer Science*, 1988.

[9] K. Ichikawa and S. Chiba. Composable user-defined operators that can express user-defined literals. In *MODULARITY*, 2014.

[10] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. *The OCaml system release 4.01 Documentation and user's manual.* Institut National de Recherche en Informatique et en Automatique, September 2013.

[11] F. Lorenzen and S. Erdweg. Modular and automated type-soundness verification for language extensions. In *ICFP*, 2013.

[12] The Coq development team. *The Coq proof assistant reference manual.* LogiCal Project, 2004. Version 8.0.

[13] C. Omar, D. Kurilova, L. Nistor, B. Chung, A. Potanin, and J. Aldrich. Safely composable type-specific languages. In *ECOOP*, 2014.

[14] C. Omar, C. Wang, and J. Aldrich. Composable and hygienic typed syntax macros. Technical Report CMU-ISR-14-113, Carnegie Mellon University.

[15] B. C. Pierce and D. N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, Jan. 2000.

[16] W. V. O. Quine. *Mathematical Logic.* Harvard University Press, Boston, MA, 1940.

[17] C. Scaffidi, A. Cypher, S. Elbaum, A. Koesnandar, J. Lin, B. Myers, and M. Shaw. Using topes to validate and reformat data in end-user programming tools. In *4th International Workshop on End-User Software Engineering*, 2008.

[18] A. Schwerdfeger and E. V. Wyk. Verifiable composition of deterministic grammars. In *PLDI*, 2009.

[19] K. Skalski, M. Moskal, and P. Olszta. Meta-programming in Nemerle. In *GPCE*, 2004.

[20] G. L. Steele. *Common LISP: the language.* Digital Press, 1990.