Composable and Hygienic Typed Syntax Macros

ABSTRACT

Syntax extension mechanisms are powerful, but ensuring that extensions are individually well-behaved and can be unambiguously composed is difficult. Recent work on *type-specific languages (TSLs)* addressed these problems in the specific setting of literal forms. We supplement TSLs with *typed syntax macros (TSMs)*, which are explicitly invoked to give meaning to delimited segments of arbitrary syntax, at both the level of terms and types. To maintain a typing discipline, we describe two flavors of term-level TSMs: synthetic TSMs specify the type of term that they elaborate to, while analytic TSMs can elaborate to 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 generate a type declaration of a specified kind and its corresponding TSL. We formally specify these mechanisms by extending the bidirectionally typed elaboration semantics previously given for TSLs, building on its hygiene mechanism and using the same internal language. Taken together, TSLs and TSMs provide significant expressive power without compromising composability, hygiene and the typing and kinding discipline of the language.

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 directly ensure that no parsing ambiguities are possible and that desugarings are semantically well-behaved. Instead, the mechanism must be tasked with maintaining several key guarantees:

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 newly introduced desugaring logic 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 statically typed language, which will be our focus in this work, maintaining a *typing discipline* is also desirable: 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 code editors.

Most prior approaches to syntax extension, discussed in Sec. 6, fail to simultaneously provide all of these guarantees. Recent work on *type-specific languages (TSLs)* makes these guarantees, but only in a limited setting: library providers can define new syntax only for introducing values of a type (i.e. *literal forms*), associating it as metadata with the type when it is declared [8]. Local type inference, specified as a bidirectional type system [9], controls which TSL is used to parse delimited literal forms containing arbitrary syntax, so TSLs are flexible, composable and maintain the typing discipline. The semantics given also guarantees hygiene. We will review and give an example of a TSL in Sec. 2.

While many forms of syntactic sugar can be implemented as TSLs, there remain situations where TSLs do not suffice: 1) only a single TSL can be associated with a type, and only when it is declared, so alternative syntactic choices, or syntax for a type not under a user's control, cannot be defined; 2) 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; and 3) types cannot themselves be declared using specialized syntax. In this paper, we introduce *typed syntax macros (TSMs)*, which supplement TSLs to support these scenarios while maintaining the strong, and we believe crucial, guarantees above.

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. Both TSLs and TSMs lever-

XXX XXX

```
1  type HTML = casetype
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    metadata = new : HasTSL
10     val parser = ~
11       start <- '<body' attributes '>' start '</body>'
12         fn atts, child => 'BodyElement(($atts, $child))'
13       start <- '<{' EXP '}>'
14         fn e => e
15       (* ... *)
16
17 let heading : HTML = H1Element({}, Text("My Heading"))
18 serve(~) (* serve : HTML -> Unit *)
19   <body id="doc1">
20     <{heading}>
21     <p>My first paragraph.</p>
22   </body>
```

Figure 1: A case type with an associated TSL.

age the same basic set of lightweight delimiters to separate syntax extensions from the host language. We supplement those previously defined with *multipart delimited forms* to support additional syntactic idioms.

We next turn to the level of 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. In doing so, we also extend the work on TSLs to handle parameterized types.

In Sec. 5, we give a minimal type-theoretic account of these mechanisms by extending the bidirectionally typed elaboration semantics for TSLs given previously by Omar et al. [8], leveraging the same underlying semantics and hygiene mechanism at the term level and introducing an analagous type-level hygiene mechanism.

Taken together, TSLs and TSMs represent what we see as a new "high water mark" in expressive power, particularly within the space of systems that 1) provide the strong guarantees described above; and 2) are rigorously specified in type theoretic terms. We more specifically compare our work to related work in Sec. 6.

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 [8], 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 concrete syntax.

An example of a type encoding the tree structure of HTML is declared in Figure 1. The named 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 can introduce a value of type `HTML` by naming a case and providing data of the type the case declares, seen on line 17.

Types declared in this way are distinguished by name (they are not simply type aliases). Wyvern supports both

```
1  type HasTSL = objtype
2    val parser : Parser(Exp)
3
4  type Parser(T) = objtype
5    def parse(ParseStream) : Result(T)
6    metadata = new : HasTSL
7      val parser = (* ... parser generator ... *)
8
9  type Result(T) = casetype
10   OK of T
11   Error of String * Location
12
13 type Exp = casetype
14   Var of ID
15   Lam of ID * Exp
16   Ap of Exp * Exp
17   Ascription of Exp * Type
18   CaseIntro of ID * Exp
19   (* ... *)
20   Spliced of ParseStream
21   metadata = new : HasTSL
22     val parser = (* ... exp quasiquotes ... *)
23
24 type Type = casetype
25   TVar of ID
26   TLam of ID * Type
27   TAp of Type * Type
28   Named of ID
29   Objtype of List(MemberDecl)
30   Casetype of List(CaseDecl)
31   Arrow of Type * Type
32   Spliced of ParseStream (* see Sec. 5 *)
33   metadata : HasTSL = new
34     val parser = (* ... type quasiquotes ... *)
```

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

named and unnamed case types, tuple types, written e.g., `HTML * HTML`, function types, written e.g., `HTML -> Unit`, and object types, which are structural (rather than class-based) and can declare fields via `val` and methods via `def`. Two object types are shown in Figure 2, described further below. Objects are introduced with `new`, which serves as a syntactic *forward reference*: it can appear once per line, at any term position. The next indented block gives values for the fields and implementations for the methods.

We also add named parameterized type constructors, e.g. `List`. Applying this type constructor to a type produces a type, e.g. `List(String)`. Types have *kind* `*`, while type constructors have arrow kind, e.g. `List` has kind `* -> *`. For concision, we write *named type* to refer to names having any kind. We assume that the definitions of standard named types like `String`, `List` and `Option` are ambiently available via the *prelude*.

All named types can be equipped with *metadata*: a value constructed at compile time and available for use by the language itself (in particular, the TSL mechanism) as well as tools. Metadata is analagous to class annotations in Java, but it can be any Wyvern value.

2.2 Type-Specific Languages (TSLs)

Introducing a value of a type like `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 compatibility. To allow for this, we associate a *type-specific language* with the `HTML` type by setting the metadata to a value of type `HasTSL`, an object type with a field `parser` of type `Parser(Exp)`.

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

Figure 3: Available delimited forms. The curly brace delimited and multipart delimited forms are novel and are shown being used in Sec. 3.

We see this TSL being used on lines 18-22 of Figure 1. On line 18, we wish to 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 for the literal. We could equivalently have used other *inline delimiters*, e.g. curly braces or single quotes, which restrict what can appear inside them, as described in Figure 3. For example, we could have written line 17 equivalently as:

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

When the type system encounters literal forms like this, it defers to the parser associated with the named type that the literal is being analyzed against, here `HTML`. For clarity in this paper, we color host language terms black and portions of TSL (and TSM) bodies (that are not spliced host language terms, see below) a unique color corresponding to the TSL or TSM being used, identified when declared.

As suggested by the declaration of `Parser` in Figure 2, the TSL is responsible for transforming a `ParseStream` based on the body to a `Result(Exp)`, which is either an `Exp` or a parse error. The type `Exp` encodes the abstract syntax of Wyvern terms, supporting static code generation.

Rather than writing a `parse` function explicitly, we make use of the fact that `Parser` has a TSL associated with it providing a static *grammar-based parser generator*. This allows us to create a `Parser(Exp)` by giving 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 top-level non-terminal. The type `Exp` also has a TSL associated with it to support *quasiquotation*: it allows terms of type `Exp` to be constructed using Wyvern's concrete syntax, extended with an unquote form `$x` that splices in the value of the variable `x` of type `Exp`.

Splicing is possible because a parser can request that some portion of the parse stream be treated as a host language term, type or variable. For example, the TSL for `HTML` uses the delimiters `<{` and `}>` to mean "splice in the enclosed term of type `HTML` here". The special cases called `Spliced` in `Exp` and `Type` capture such spliced terms; tracking which portions of a parse stream were spliced terms is the basis of the hygiene mechanism, which we return to in Sec. 5. The parser generator provides the non-terminals `EXP`, `ID` and `TYPE`, which generate these spliced forms internally.

3. TERM-LEVEL TSMs

In this section, we will give examples of term-level typed syntax macros in Wyvern to illustrate how they are defined and can be used in situations where TSLs are not suitable.

```
syntax simpleHTML => HTML = ~ (* : Parser(Exp) *)
  start <- '>body'= attributes> start>
    fn atts, child => 'BodyElement(($atts, $child))'
  start <- '<'= EXP>
    fn e => e
  (* ... *)
let heading = simpleHTML '>h1 My Heading'
serve(simpleHTML ~)
  >body[id="doc1"]
    < heading
    >p My first paragraph
```

Figure 4: A synthetic TSM providing alternative syntax for the `HTML` type in Figure 1. The programs are semantically identical.

3.1 Synthetic TSMs

TSMs are defined using the `syntax` keyword. Figure 4 shows a synthetic TSM, `simpleHTML`, being defined and used. The annotation on the first line indicates that valid uses of the TSM must elaborate to a term that synthesizes the type `HTML`. Like defining a TSL, defining a TSM requires defining a parser, i.e. a statically-evaluated value of type `Parser(Exp)`. For the purposes of exposition, we include type annotations that are not strictly needed in comments.

Applying a TSL is similar to function applying, but rather than using parenthesis, the name of the TSL is followed by a delimited form (Figure 3). The body of the delimited form is parsed according to the definition of the TSM. Note that we do not here address namespacing issues, as standard techniques can be used (e.g. using URIs as in Java).

Notice here that on line 7, we do not need a type annotation on `heading` because `simpleHTML` is synthetic. On lines 8-11, we use the same forward referenced delimited form introduced in the work on TSLs to avoid syntactic clashes between explicit delimiters and the extended syntax. The only difference here is the addition of the `simpleHTML` "keyword", which indicates to the type system that the TSM should be used rather than the TSL for `HTML`. As a result, both variants of syntax can straightforwardly be used in the same program, so synthetic TSMs address the issue of defining more than one possible syntax for a type that either has a TSL already, or a type which a user cannot modify.

We again define the parser for `simpleHTML` by using the parser generator implemented as a TSL for `Parser`. Here, we are defining an alternative layout-sensitive syntax for HTML that is more concise than the conventional one by way of an *Adams grammar*, which supports declarative specifications of layout-sensitive grammars by using *layout constraints* within productions [1]. Here, the suffix `=` indicates that the left-most column (on any line) occupied by the annotated terminal or non-terminal must occur at the same column as the parent production and `>` indicates that it must be further indented. More detail on Adams grammars and this syntax for HTML can be found in [8].

3.2 Analytic TSMs

Some idioms may be valid at many types. As perhaps the simplest example, consider a case type encoding booleans, `Bool`, shown in Figure 5. Using explicit case analysis on booleans is often unnecessarily verbose, so we introduce the

```
1  type Bool = casetype
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 {simpleHTML '>h1 Not OK!'}
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.

more idiomatic `if` construct. Rather than having to build this in to the language, however, we can implement it as an analytic TSM. These are distinguished from synthetic TSMs by the absence of a type annotation, because the type of an `if` expression is determined by its branches.

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 identifier (e.g. `else`). The body is generated by concatenating the bodies of the parts and inserting a special boundary character outside the normal character set between them. 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]` generate an equivalent body, `e1·else·e2` where · is the boundary.

For the branches in our example, we chose to use curly brace delimiters, which differ slightly from the previous work on TSLs so as to be specialized for situations where the body consists of a single spliced term. 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. Wyvern programmers would be expected to be comfortable with forward references, so this is more idiomatic Wyvern code. Were, for example, square brackets used, then we would need to write the example as follows:

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

An analytic TSM can only be used in a position where the type is otherwise known, e.g. due to the return type annotation on line 10. 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 allow analytic TSMs to be used in synthetic positions. For example, we might permit a post-fix asterisk. 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 [7]):

```
def testIf3(ok : Bool) (* no return type annotation *)
  if* [ok] {simpleHTML ~} else {simpleHTML '>h1 Not OK!'}
    >h1 Everything is OK!
```

```
type 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 scheme.

4. TYPE-LEVEL TSMs

We now turn our attention to TSMs at the level of named type declarations. Our example is a simple object-relational mapping (ORM) syntax, shown being used in Figure 6. An 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 based on the schema during compilation, using a language-integrated mechanism.

More specifically, 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, named types in Wyvern can be declared with metadata. A type-level TSM can generate not just the underlying type for a named type, but also its metadata. Because TSLs are defined using this metadata mechanism, this implies that type-level TSMs can generate TSLs. Here, the TSL that the `schema` TSM generates is shown being used on lines 5-10 of Figure 8 to create a value of type `EmployeesDB`. Only the per-database settings need to be provided.

The definition of `schema` is shown in Figure 8. For a `syntax` declaration to define a type-level TSM, there must be a specification of the *kind* of type-level term that will be generated, here `*` because the type takes no parameters, as well as the type of metadata that will be generated, here `HasTSL`. The elaboration is then defined by a parser that produces a pair consisting of a reified type-level term (`Type` is analagous to `Exp`, cf. Figure 2) and the metadata value.

Here, we generate the type by using type quasiquotation, unquoting using `${...}` to generate field and method declarations by mapping over the column specifications in the provided schema (using the same color for `Type` and `MemberDecl` for simplicity). We assume a mapping from SQL types to Wyvern types, `ty_from_sqlty`, not shown. Starting on line 21, we then generate the metadata value,

```
type EmployeesDB = objtype
  type 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)
  metadata = new : HasTSL
    val parser = (* ... 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.

which defines a TSL as described in Section 2. Note that the TSL implements the `getByC` methods by mapping over the column specification provided to the type-level TSM. In other words, the implementation of the type generated on lines 4-20 is filled in by the TSL generated on lines 21-38.

The metadata mechanism in Wyvern supports more than just TSLs. For example, a documentation generator might look for a `doc` field in the metadata. Our type-level TSM only generates the TSL definition. To support extending the metadata generated by a type-level TSM further after it is invoked, the mechanism supports a *metadata transformation* when a type is declared using a type-level TSM:

```
type EmployeesDB2 = (schema ~
  ...
) metadata fn original_md (* : HasTSL *) => new
  val parser = original_md.parser
  val doc = "ORM for Employees table in database."
```

The metadata type declared by the type-level TSM determines the input type of the transformation (here, `HasTSL`).

5. FORMAL SYNTAX AND SEMANTICS

We will now give a formal type theoretic treatment of the mechanisms described thusfar, building directly upon the calculus for TSLs in Wyvern presented previously by Omar et al. [8]. The abstract syntax corresponding to the concrete syntax we introduced above is shown in Figures 9 and 10. A *program*, ρ, consists of a series of declarations, d, followed by a single external term, e. In practice, the declarations would be imported from separate packages, but we omit details of the package system here for simplicity. The top-level judgement in the system is the compilation judgement:

$$\frac{d \sim (\Psi;\Theta) \qquad \emptyset;\emptyset \vdash^{\Psi}_{\Theta_0\Theta} e \rightsquigarrow i \Rightarrow \tau}{d; e \sim (\Psi;\Theta) \rightsquigarrow i : \tau} \text{(compile)}$$

A series of declarations, d, consists of TSM declarations and named type declarations. The judgement $d \sim (\Psi, \Theta)$, defined in Figure 12, generates a corresponding TSM context, Ψ, and named type context, Θ, from d. The syntax for these contexts is given in Figure 11. We refer to the type declarations in the prelude, some of which were shown in Fig. 2, by using the named type context Θ_0 in our rules.

```
syntax schema :: * with metadata : HasTSL = ~ (* : Parser
    (Type * HasTSL) *)
  start <- columns
    fn cols (* : List(Bool*Label*Type), ln. 40–46 *) =>
      let ty = ~ : Type (* not recursive, so _ ignored *)
        objtype
          type Entry = objtype
            ${(* the members of Entry are generated by
                 mapping over the columns *)
              map(cols, fn (primary, lbl, ty) => ~)
                val $lbl :  $ty
              }
            val connection :  URL
            val username :  String
            val password :  String
            (* ...  *)
            ${(* the getByC method signatures are also
                 generated in this way *)
              map(cols, fn (primary, lbl, ty) =>
                let rty : Type = if [primary] {
                  'Option(Entry)'} else {'List(Entry)'}
                ~)
                  def getBy$lbl($ty) :  $rty
              }
        let md = new : HasTSL
          val parser = ~ (* : Parser(Exp) *)
            start <- ("connect to"= EXP>
                      "table"= EXP>
                      "username"= EXP>
                      "password"= EXP>)
              fn url, un, pw, table => ~
                new
                  val connection = $url
                  val username = $un
                  val password = $pw
                  (* ...  *)
                  ${(* like their declarations above, the
                       getByC method implementations also
                       map over the columns *)
                    map(cols, fn (primary, lbl, ty) => ~)
                      def getBy$lbl(x)
                        (* send appropriate query *)
                    }
        (ty, md)
  column <- "*"?  ID ID
    fn primary, lbl, sqlty =>
      (primary, lbl, ty_from_sqlty(sqlty))
  columns <- column
    fn column => Cons(column, Nil)
  columns <- column= columns=
    fn column, columns => Cons(column, columns)
```

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

Judgements $\Delta;\Gamma \vdash^{\Phi}_{\Theta} e \rightsquigarrow i \Rightarrow \tau$ and $\Delta;\Gamma \vdash^{\Phi}_{\Theta} e \rightsquigarrow i \Leftarrow \tau$ can be read "under kinding context Δ, typing context Γ, named type context Θ and TSM context Φ, external term e elaborates to internal term i and (synthesizes / analyzes against) type τ". This is a *bidirectionally typed elaboration semantics*, used to elaborate TSL literals and TSM applications, which appear only in the external language, to an internal language, which includes only the core operations in the language. The semantics follows that given in [8], so we do not repeat the rules for the core operations or TSL literals. The rules for the only new form, $\mathbf{eaptsm}[s, body]$, are given in Figure 13 and described below.

5.1 Term-Level TSMs

The rule (D-syntsm) shows how a synthetic TSM named s, synthesizing type τ and implemented by e_{tsm} is declared. The rule first recursively checks the preceding declarations, then ensures that no other TSM named s was declared (we treat contexts as finite mappings and write, e.g., $\text{dom}(\Psi)$ to be the domain of Ψ). Then, it checks that τ is a closed type

Abstract Forms	**Concrete Forms**
Programs ρ ::= $d; e$	
Declarations d ::=	
$\emptyset$	
$d; \mathbf{syntsm}(s, \tau, e)$	`syntax` s `:` τ `=` e
$d; \mathbf{anatsm}(s, e)$	`syntax` s `=` e
$d; \mathbf{tytsm}(s, \kappa, \tau, e)$	`syntax` s `::` κ `with metadata:` τ `=` e
$d; \mathbf{tydecl}(\mathtt{T}, \tau, e)$	`type T =` τ `metadata =` e
$d; \mathbf{tyaptsm}(\mathtt{T}, s, body, e)$	`type T =` s $dform$ `metadata` e
External Terms e ::= ...	
$\mathbf{lit}[body]$	$dform$
$\mathbf{eaptsm}[s, body]$	s $dform$

Translational Terms $\hat{e}$ ::= ... | $\mathbf{spliced}[e]$
Internal Terms i ::= ...

Figure 9: Abstract and concrete forms for declarations and terms. Metavariable s ranges over TSM names, `T` over type names, $dform$ over delimited forms, per Figure 3, and $body$ over their bodies. Translational and internal terms are used in the semantics only. Elided forms are given in [8].

Kinds κ ::= $\star \mid \kappa \to \kappa$
Types τ ::= $\mathtt{T} \mid \tau \to \tau \mid \mathbf{objtype}[\omega] \mid \mathbf{casetype}[\chi]$
$\mid \; t \mid \lambda[\kappa](t.\tau) \mid \tau(\tau)$
Translational Types $\hat{\tau}$ ::= ... | $\mathbf{spliced}[\tau]$

Figure 10: Syntax for types and kinds. Metavariable t ranges over type variables. Object type and case type declarations ω and χ are taken from [8].

by checking that it's kind is $\star$ in an empty kinding context. Kinding contexts are simply mappings from type variables to kinds and the key kinding rules are shown in Figure 14. We will return to them when discussing type declarations below. Finally, e_{tsm} is analyzed against `Parser(Exp)`, defined in the prelude. It must be a closed term, so the kinding and typing contexts are empty (it can use the named types and TSMs declared previously, however). Once these checks are complete, the definition of s is added to Ψ. The rule for analytic TSM declarations, (D-anatsm), is nearly identical, differing only in that no kind check is needed.

Term-level TSM application is captured by the abstract form $\mathbf{eaptsm}[s, body]$, where s is the name of the TSM and $body$ is the body of the delimited form, per Sec. 3. The rule (T-syn) shows how typing and elaboration for a synthetic TSM proceeds. First, the definition of s is extracted from Ψ. Then, a parse stream is constructed on the basis of $body$. We assume the relation $\mathsf{parsestream}(body) = i_{ps}$ is defined such that i_{ps} is a closed term of type `ParseStream`. Then, the $parse$ method of the TSM implementation is invoked with the parse stream. The judgement $i \Downarrow i'$ captures evaluation of i to a value, i'. Our internal language is identical to that in [8], so we omit the rules. As suggested by the declarations in Figure 2, the result is either a parse error or a valid parse, written $OK(i_{exp})$, where i_{exp} is a value of type `Exp`. Here, we simply leave the error case undefined – the typing judgement cannot be derived if there is a parse error.

The *dereification judgement* $i \uparrow \hat{e}$, defined in [8], takes a value of type `Exp` to a corresponding *translational term*, $\hat{e}$. Translational terms mirror external terms but include an additional form, $\mathbf{spliced}[e]$, which captures portions of the parse stream parsed as a spliced term. The case `Spliced` of case type `Exp`, which takes a (portion of) a parse stream, dereifies to this form. This permits us to ensure that only spliced portions of parse streams can refer to variables in the surrounding scope (and no others), ensuring that hygiene is maintained.

This is technically accomplished by the judgements

$$\Delta_{out}; \Gamma_{out}; \Delta; \Gamma \vdash^{\Psi}_{\Theta} \hat{e} \rightsquigarrow i \Rightarrow (\Leftarrow) \tau$$

which can be read "under outer typing and kinding contexts Δ_{out} and Γ_{out} and inner typing and kinding contexts Δ and Γ, $\hat{e}$ elaborates to internal term i and (synthesizes/-analyzes against) type τ". These judgements behave identically to the corresponding judgements for external terms, using the inner typing and kinding contexts, until a term of the form $\mathbf{spliced}[e]$ is encountered. The outer contexts are then used. The relevant rule can be found in [8], and an analagous type-level rule will be shown below. In the rules for TSLs and TSMs, (T-syn) and (T-ana), the inner contexts begin empty so only variables inside spliced terms can refer to outer variables. A `parse` function that generated, for example, `Var('x')`, would not typecheck, because it captures a variable that it cannot know exists. An occurrence of a variable `x` inside a spliced portion (e.g. between `<{` and `}>` when using the `HTML` TSL) would be checked in the outer context and thus be acceptable. Parse streams cannot be created manually, so this guarantee is strict.

In the rule (T-syn), the type that $\hat{e}$ is being analyzed against is determined by the definition of the TSM. The rule (T-ana) is essentially identical, but the type is determined by the type that the whole application is being analyzed against instead, consistent with the descriptions in Sec. 3.

TSM Contexts
$\Psi ::= \emptyset \mid \Psi, s[\mathbf{ty}(\kappa, \tau, i)] \mid \Psi, s[\mathbf{syn}(\tau, i)] \mid \Psi, s[\mathbf{ana}(i)]$
Named Type Contexts $\Theta ::= \emptyset \mid \Theta, \mathtt{T}[\tau :: \kappa, i : \tau]$
Typing Contexts $\Gamma ::= \emptyset \mid \Gamma, x : \tau$
Kinding Contexts $\Delta ::= \emptyset \mid \Delta, t :: \kappa$

Figure 11: Syntax for contexts.

$\boxed{\Delta; \Gamma \vdash^{\Psi}_{\Theta} e \rightsquigarrow i \Rightarrow \tau}$

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

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

Figure 13: Statics for Keyword Invocation

5.2 Type Declarations

The rule (D-tydecl) shows how explicit named type declarations (those which do not apply a type-level TSM) work. First, the preceding declarations are processed and we ensure that no other type named T was declared. Then, we

$\boxed{d \sim (\Psi; \Theta)}$

$$\frac{}{\emptyset \sim (\emptyset; \emptyset)}\ \text{(D-empty)}$$

$$\frac{d \sim (\Psi; \Theta) \qquad s \notin \mathrm{dom}(\Psi) \qquad \emptyset \vdash_{\Theta_0\Theta} \tau :: \star \qquad \emptyset; \emptyset \vdash^{\Psi}_{\Theta_0\Theta} e_{tsm} \rightsquigarrow i_{tsm} \Leftarrow \texttt{Parser(Exp)}}{d; \mathbf{syntsm}(s, \tau, e_{tsm}) \sim (\Psi, s[\mathbf{syn}(\tau, i_{tsm})]; \Theta)}\ \text{(D-syntsm)}$$

$$\frac{d \sim (\Psi; \Theta) \qquad s \notin \mathrm{dom}(\Psi) \qquad \emptyset; \emptyset \vdash^{\Psi}_{\Theta_0\Theta} e_{tsm} \rightsquigarrow i_{tsm} \Leftarrow \texttt{Parser(Exp)}}{d; \mathbf{anatsm}(s, e_{tsm}) \sim (\Psi, s[\mathbf{ana}(i_{tsm})]; \Theta)}\ \text{(D-anatsm)}$$

$$\frac{d \sim (\Psi; \Theta) \qquad s \notin \mathrm{dom}(\Psi) \qquad \emptyset \vdash_{\Theta_0\Theta} \tau_{md} :: \star \qquad \emptyset; \emptyset \vdash^{\Psi}_{\Theta_0\Theta} e_{tsm} \rightsquigarrow i_{tsm} \Leftarrow \texttt{Parser(Type} \times \tau_{md}\texttt{)}}{d; \mathbf{tytsm}(s, \kappa, \tau_{md}, e_{tsm}) \sim (\Psi, s[\mathbf{ty}(\kappa, \tau_{md}, i_{tsm})]; \Theta)}\ \text{(D-tytsm)}$$

$$\frac{d \sim (\Psi; \Theta) \qquad \mathtt{T} \notin \mathrm{dom}(\Theta_0\Theta) \qquad \emptyset \vdash_{\Theta_0\Theta} \tau :: \kappa \to \kappa \qquad \emptyset; \emptyset \vdash^{\Psi}_{\Theta_0\Theta, \mathtt{T}[\tau(\mathtt{T}) :: \kappa, -]} e_{md} \rightsquigarrow i_{md} \Rightarrow \tau_{md}}{d; \mathbf{tydecl}(\mathtt{T}, \tau, e) \sim (\Psi; \Theta, \mathtt{T}[\tau(\mathtt{T}) :: \kappa, i_{md} : \tau_{md}])}\ \text{(D-tydecl)}$$

$$\frac{\begin{array}{c} d \sim (\Psi; \Theta) \qquad \mathtt{T} \notin \mathrm{dom}(\Theta_0\Theta) \qquad s[\mathbf{ty}(\kappa, \tau_{md}, i_{tsm})] \in \Psi \\ \mathsf{parsestream}(body) = i_{ps} \qquad i_{tsm}.parse(i_{ps}) \Downarrow OK((i_{type}, i_{md})) \qquad i_{type} \uparrow \hat{\tau} \qquad \emptyset; \emptyset \vdash_{\Theta\Theta'} \hat{\tau} \rightsquigarrow \tau :: \kappa \to \kappa \\ \emptyset; \emptyset \vdash^{\Psi}_{\Theta_0\Theta, \mathtt{T}[\tau(\mathtt{T}) :: \kappa, -]} e_{mdx} \rightsquigarrow i_{mdx} \Rightarrow \tau_{md} \to \tau'_{md} \qquad i_{mdx}(i_{md}) \Downarrow i'_{md} \end{array}}{d; \mathbf{tyaptsm}(\mathtt{T}, s, body, e_{mdx}) \sim (\Psi; \Theta, \mathtt{T}[\tau(\mathtt{T}) :: \kappa, i'_{md} : \tau'_{md}])}\ \text{(D-aptsm)}$$

Figure 12: Declaration checking and elaboration of type-level TSMs.

$\boxed{\Delta \vdash_\Theta \tau :: \kappa}$

$$\frac{t :: \kappa \in \Delta}{\Delta \vdash_\Theta t :: \kappa} \qquad \frac{\Delta \vdash_\Theta \tau_1 :: \kappa \to \kappa' \qquad \Delta \vdash_\Theta \tau_2 :: \kappa}{\Delta \vdash_\Theta \tau_1(\tau_2) :: \kappa'}$$

$$\frac{\Delta, t :: \kappa \vdash_\Theta \tau :: \kappa'}{\Delta \vdash_\Theta \lambda[\kappa](t.\tau) :: \kappa \to \kappa'} \qquad \frac{T[\tau :: \kappa, i_{md} : \tau_{md}] \in \Theta}{\Delta \vdash_\Theta \mathtt{T} :: \kappa}$$

Figure 14: Kinding (object and case types follow [8])

$\boxed{\Delta; \Delta \vdash_\Theta \hat{\tau} \rightsquigarrow \tau :: \kappa}$

$$\frac{\Delta_{out} \vdash_\Theta \tau :: \kappa}{\Delta_{out}; \Delta \vdash_\Theta \mathbf{spliced}[\tau] \rightsquigarrow \tau :: \kappa} \qquad \frac{t :: \kappa \in \Delta}{\Delta_{out}; \Delta \vdash_\Theta t :: \kappa}$$

$$\frac{\Delta_{out}; \Delta, t :: \kappa \vdash_\Theta \hat{\tau} \rightsquigarrow \tau :: \kappa'}{\Delta_{out}; \Delta \vdash_\Theta \lambda[\kappa](t.\hat{\tau}) \rightsquigarrow \lambda[\kappa](t.\tau) :: \kappa \to \kappa'}$$

Figure 15: Translational Kinding. Remaining rules are directly analagous to those for τ in Figure 14.

need that the type τ has arrow kind $\kappa \to \kappa$ where κ is the kind of type being declared, e.g. $\kappa = \star \to \star$ for `List`. The reason for this is to support recursive named types, e.g.

```
type List(T) = casetype
  Nil
  Cons of T * List(T)
```

desugars to a type-level function taking in a self reference before returning a type-level function accepting the type parameter and returning the case type being declared:

```
type List = tyfn(List::* -> *) => tyfn(T::*) => casetype
  Nil
  Cons of T * List(T)
```

Before being added to the named type context, the named type constructor `List` is substituted for the type variable *List* via type-level function application, $\tau(\mathtt{T})$, as can be seen in the conclusion of the rule. Non-recursive types can simply ignore the "self reference" argument.

We do not formalize mutually recursive types here, though they follow the same pattern (taking n arguments rather than just one). The full rules can be found in the appendix. Note also that we do not explicitly check for cyclic type definitions. Standard syntactic constraints can be imposed to rule these out.

The final premise of (D-tydecl) synthesizes a type for the metadata. Metadata can use the type declaration, but not recursively refer to its own metadata yet so we write a dash for a dummy metadata value.

Figure 14 shows that named types have the kind of their underlying type-level term. Parameterized types can thus be applied to parameters like type-level functions.

5.3 Type-Level TSMs

Rule (D-tytsm) declares a type-level TSM s that generates a type of kind κ with metadata of type τ_{md} and an implementation e_{tsm}. The metadata type is checked to ensure that it is a type and the implementation is checked against type $\texttt{Parser(Type} \times \tau_{md}\texttt{)}$, consistent with the explanation in Section 4. Note that we assume product types can be encoded as object types for simplicity in our calculus. This information is recorded in the TSM context, Ψ.

Rule (D-aptsm) shows how type-level TSMs are applied. If the result of the `parse` method is $OK((i_{type}, i_{md}))$, then we dereify i_{type} to a *translational type*, $\hat{\tau}$. These are analagous to translational terms, $\hat{e}$, and serve to ensure hygiene at the level of types. The judgement $\Delta_{out}; \Delta \vdash_\Theta \hat{\tau} \rightsquigarrow \tau :: \kappa$ can be read "under outer kinding context Δ_{out}, inner kinding context Δ and named type context Θ, translational type $\hat{\tau}$ elaborates to type τ at kind κ". The syntax of translational types mirrors that of types, again with the addition

of a form capturing spliced type-level terms arising from the parse stream, **spliced**[τ]. The rules related to type variables and spliced forms are shown in Figure 15. Only spliced types are checked under the outer context.

As with type declarations, in (D-aptsm) we need that $\hat{\tau}$ has kind $\kappa \to \kappa$ to support recursive types (in the example in Sec. 4, we omitted this because in practice, the mechanism inserts it by default for non-recursive types). The outer kinding context here is initially empty because we do not formalize nested type declarations for simplicity, but in practice it would be the kinding context coming from any outer declarations.

The remaining two premises of (D-aptsm) check that the metadata transformation is a function of the right type and invoke it to produce the final metadata (per Sec. 4). An identity function would be generated automatically when one is not explicitly provided by the user.

5.4 Metatheory

The main theorems guaranteeing that the language is well-behaved are essentially identical to those in [8]. The key theorem states that well-typed external terms translate to internal terms of the same type. Combined with type safety of the IL, this constitutes type safety for the language.

THEOREM 1 (EXTERNAL TYPE PRESERVATION). *If* $\vdash \Theta$ *and* $\vdash_\Theta \Psi$ *and* $\vdash_\Theta \Gamma$ *and* $\Gamma \vdash^{\Psi}_{\Theta} e \rightsquigarrow i \Leftarrow \tau$ *or* $\Gamma \vdash^{\Psi}_{\Theta} e \rightsquigarrow i \Rightarrow \tau$ *then* $\Gamma \vdash_\Theta i \Leftarrow \tau$.

We define context well-formedness judgements and prove the new cases of this theorem in the accompanying appendix, along with several other lemmas and theorems.

6. RELATED WORK

Unlike other work on library-integrated syntax extension mechanisms, e.g. SugarJ and its subsequent variations [4], and language-external mechanisms like Camlp4 [6], we do not permit the syntax of the base language to be extended directly. Instead, we build on the same delimited forms used for type-specific languages [8]. This guarantees composability – any combination of libraries can be imported and used together, without "link-time" ambiguities, because different extensions can only interact via the host language (using splicing). We retain the layout delimited forward referenced form and we introduce two variations on the delimiters used by [8], multipart delimited forms and a curly-brace delimited form that allow escape of forward references. Ichikawa and Chiba recently described *ProteaJ*, which are similar to TSLs and suffer from the same deficiencies addressed here, as well as the problem of conflicts with the base language because delimiters are not used [5].

Schwerdfeger and Van Wyk have shown a composable analysis for syntax specified using an LR parser generator with a context-aware scanner [10]. 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 [1]) 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).

None of these mechanisms supports a typing discipline directly, though Lorenzen and Erdweg have described a mechanism for proving the admissibility of derived typing rules for syntax extensions [7].

Macro systems have a long history in the LISP family of languages. These typically only permit reinterpretting 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 [12]. Nemerle provides a similar facility that does use delimiters [11]. These do not consider issues related to the typing discipline. Macro systems that do consider the typing discipline, e.g. in Scala [2], do not support syntax extension. In Scala, *black box macros* are like synthetic TSMs: they give a type signature. Analytic TSMs do not directly have an analog, but they can be seen as a restriction on the use of *white box macros* (disabled by default) to analytic positions.

Concerns about hygiene have been well-studied in these communities, e.g. in Scheme [3]. Our formalization of the hygiene mechanism is 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 consider hygiene at both the term and type level.

7. REFERENCES

[1] M. D. Adams. Principled parsing for indentation-sensitive languages: Revisiting Landin's offside rule. In *POPL*, 2013.

[2] 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*, 2013.

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

[4] S. Erdweg and F. Rieger. A framework for extensible languages. In *GPCE '13*, pages 3–12. ACM, 2013.

[5] K. Ichikawa and S. Chiba. Composable user-defined operators that can express user-defined literals. In *Modularity*, pages 13–24. ACM, 2014.

[6] 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.

[7] F. Lorenzen and S. Erdweg. Modular and automated type-soundness verification for language extensions. In *ICFP*, pages 331–342. ACM, 2013.

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

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

[10] A. Schwerdfeger and E. V. Wyk. Verifiable composition of deterministic grammars. In *PLDI*, pages 199–210. ACM, 2009.

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

[12] G. L. Steele. *Common LISP: the language*. Digital press, 1990.

APPENDIX

The details below are not necessary to understand the main contributions of the paper, but may be useful for readers interested in implementing or expanding upon our work. We also include more complete proofs and proof sketches of the key metatheoretic properties of the language below. These will be available as a technical report alongside the final version of the paper, but we include the details here for the convenience of reviewers (we received permission from the PC chair).

A. SYNTAX & TYPE CHECKING

In the main body of the paper, we do not describe support for mutually recursive type declarations for concision. We also do not show how to access a TSM definition programatically at run-time, which can be useful to explicitly call the same parser as is used by the typechecker for TSM invocations. We describe the full syntax below.

A.1 Syntax

$$
\begin{array}{rcl}
\rho & ::= & d; e \\
d & ::= & \emptyset \\
 & | & d; \mathbf{syntsm}(s, \tau, e) \\
 & | & d; \mathbf{anatsm}(s, e) \\
 & | & d; \mathbf{tytsm}(s, n, \kappa, \tau, e) \\
 & | & d; \mathbf{mrectydecl}(\theta) \\
\theta & ::= & \emptyset \\
 & | & \theta, \mathbf{tydecl}(\mathtt{T}, \tau, e) \\
 & | & \theta, \mathbf{tyaptsm}(\mathtt{T}, s, body, e) \\
\tau & ::= & \mathtt{T} \\
 & | & \mathbf{objtype}[\omega] \\
 & | & \mathbf{casetype}[\chi] \\
 & | & \tau \to \tau \\
 & | & t \\
 & | & \lambda[\kappa](t.\tau) \\
 & | & \forall(t.\tau) \\
 & | & \tau(\tau) \\
 & | & \tau \times \tau \\
\omega & ::= & \emptyset \\
 & | & l[\mathbf{val}, \tau]; \omega \\
 & | & l[\mathbf{def}, \tau]; \omega \\
\chi & ::= & \emptyset \\
 & | & C[\tau]; \chi \\
e & ::= & x \\
 & | & \mathbf{easc}[\tau](e) \\
 & | & \mathbf{elet}(e; x.e) \\
 & | & \mathbf{elam}(x.e) \\
 & | & \mathbf{eap}(e; e) \\
 & | & \mathbf{enew}(m) \\
 & | & \mathbf{eprj}[l](e) \\
 & | & \mathbf{einj}[C](e) \\
 & | & \mathbf{ecase}(e)\{r\} \\
 & | & \mathbf{etoast}(e) \\
 & | & \mathbf{emetadata}[\mathtt{T}] \\
 & | & \mathbf{etsmdef}[s] \\
 & | & \mathbf{lit}[body] \\
 & | & \mathbf{eaptsm}[s, body] \\
m & ::= & \emptyset \\
 & | & \mathbf{eval}[l](e); m \\
 & | & \mathbf{edef}[l](x.e); m \\
r & ::= & \emptyset \\
 & | & \mathbf{erule}[C](x.e); r
\end{array}
\qquad
\begin{array}{rcl}
\kappa & ::= & \star \\
 & | & \kappa \to \kappa \\
n & ::= & 1 \mid n+1
\end{array}
$$

$$
\begin{array}{rcl}
\hat{\tau} & ::= & \mathtt{T} \\
 & | & \mathbf{objtype}[\omega] \\
 & | & \mathbf{casetype}[\chi] \\
 & | & \hat{\tau} \to \hat{\tau} \\
 & | & t \\
 & | & \lambda[\kappa](t.\hat{\tau}) \\
 & | & \forall(t.\hat{\tau}) \\
 & | & \hat{\tau}(\hat{\tau}) \\
 & | & \hat{\tau} \times \hat{\tau} \\
 & | & \mathbf{spliced}[\tau] \\
\hat{\omega} & :: & \emptyset \\
 & | & l[\mathbf{val}, \hat{\tau}]; \hat{\omega} \\
 & | & l[\mathbf{def}, \hat{\tau}]; \hat{\omega} \\
\hat{\chi} & ::= & \emptyset \\
 & | & C[\hat{\tau}]; \hat{\chi} \\
\hat{e} & ::= & x \\
 & | & \mathbf{hasc}[\hat{\tau}](\hat{e}) \\
 & | & \mathbf{hlet}(\hat{e}; x.\hat{e}) \\
 & | & \mathbf{hlam}(x.\hat{e}) \\
 & | & \mathbf{hap}(\hat{e}; \hat{e}) \\
 & | & \mathbf{hnew}(\hat{m}) \\
 & | & \mathbf{hprj}[l](\hat{e}) \\
 & | & \mathbf{hinj}[C](\hat{e}) \\
 & | & \mathbf{hcase}(\hat{e})\{\hat{r}\} \\
 & | & \mathbf{htoast}(\hat{e}) \\
 & | & \mathbf{hmetadata}[\mathtt{T}] \\
 & | & \mathbf{htsmdef}[s] \\
 & | & \mathbf{spliced}[e] \\
\hat{m} & ::= & \emptyset \\
 & | & \mathbf{hval}[l](\hat{e}); \hat{m} \\
 & | & \mathbf{hdef}[l](x.\hat{e}); \hat{m} \\
\hat{r} & ::= & \emptyset \\
 & | & \mathbf{hrule}[C](x.\hat{e}); \hat{r}
\end{array}
\qquad
\begin{array}{rcl}
i & ::= & x \\
 & | & \mathbf{iasc}[\tau](\dot{e}) \\
 & | & \mathbf{ilet}(i; x.i) \\
 & | & \mathbf{ilam}(x.i) \\
 & | & \mathbf{iap}(i; i) \\
 & | & \mathbf{inew}(\dot{m}) \\
 & | & \mathbf{iprj}[l](i) \\
 & | & \mathbf{iinj}[C](i) \\
 & | & \mathbf{icase}(i)\{\dot{r}\} \\
 & | & \mathbf{itoast}[i] \\
\dot{m} & ::= & \emptyset \\
 & | & \mathbf{ival}[l](i); \dot{m} \\
 & | & \mathbf{idef}[l](x.i); \dot{m} \\
\dot{r} & ::= & \emptyset \\
 & | & \mathbf{irule}[C](x.i); \dot{r}
\end{array}
$$

In the type checking and elaboration rules, we use certain concrete forms for simplicity and readability consideration. For example, field projection out of an object is written abstractly $\mathbf{iprj}[l](i)$, but we present it in its concrete form $i.l$ in the rules. Function application, written $i(i)$, is abstractly written $\mathbf{iap}(i; i)$. Introducing a value of a case type is written $[C](i)$, but abstractly is written $\mathbf{iinj}[C](i)$ ("injection", following standard terminology for sum types). Note also that in [8], named types were written $\mathbf{named}[T]$ but for concision here we simply use typewriter font, T. We also include polymorphic function

types in the syntax for completeness, assuming a standard mechanism supporting higher-rank polymorphism so that we do not need explicit type abstraction and application at the term level, though we do not formalize it here.

A.2 Contexts

$$\begin{array}{rrcl}
\text{TSM Context} & \Psi & ::= & \emptyset \\
& & \mid & \Psi; s[\mathbf{ty}(n, \kappa, \tau, i)] \\
& & \mid & \Psi; s[\mathbf{syn}(\tau, i)] \\
& & \mid & \Psi; s[\mathbf{ana}(i)] \\
\text{Named Type Context} & \Theta & ::= & \emptyset \\
& & \mid & \Theta, \mathtt{T}[\tau :: \kappa, i : \tau] \\
\text{Typing Context} & \Gamma & ::= & \emptyset \\
& & \mid & \Gamma, x : \tau \\
\text{Kinding Context} & \Delta & ::= & \emptyset \\
& & \mid & \Delta, t :: \kappa
\end{array}$$

A.3 Type Checking and Elaboration

$\boxed{\rho \sim (\Psi; \Theta) \rightsquigarrow i : \tau}$

$$\frac{d \sim (\Psi; \Theta) \qquad \emptyset; \emptyset \vdash^{\Psi}_{\Theta_0 \Theta} e \rightsquigarrow i \Rightarrow \tau}{d; e \sim (\Psi; \Theta) \rightsquigarrow i : \tau} \text{ (compile)}$$

$\boxed{d \sim (\Psi; \Theta)}$

$$\frac{}{\emptyset \sim (\emptyset; \emptyset)} \text{ (D-empty)}$$

$$\frac{\begin{array}{c} d \sim (\Psi; \Theta) \qquad s \notin \mathrm{dom}(\Psi) \qquad \emptyset \vdash_{\Theta_0 \Theta} \tau :: \star \\ \emptyset; \emptyset \vdash^{\Psi}_{\Theta_0 \Theta} e_{tsm} \rightsquigarrow i_{tsm} \Leftarrow \mathtt{Parser}(\mathtt{Exp}) \end{array}}{d; \mathbf{syntsm}(s, \tau, e_{tsm}) \sim (\Psi, s[\mathbf{syn}(\tau, i_{tsm})]; \Theta)} \text{ (D-syntsm)}$$

$$\frac{d \sim (\Psi; \Theta) \qquad s \notin \mathrm{dom}(\Psi) \qquad \emptyset; \emptyset \vdash^{\Psi}_{\Theta_0 \Theta} e_{tsm} \rightsquigarrow i_{tsm} \Leftarrow \mathtt{Parser}(\mathtt{Exp})}{d; \mathbf{anatsm}(s, e_{tsm}) \sim (\Psi, s[\mathbf{ana}(i_{tsm})]; \Theta)} \text{ (D-anatsm)}$$

$$\frac{\begin{array}{c} d \sim (\Psi; \Theta) \qquad s \notin \mathrm{dom}(\Psi) \qquad \emptyset \vdash_{\Theta_0 \Theta} \tau_{md} :: \star \\ \emptyset; \emptyset \vdash^{\Psi}_{\Theta_0 \Theta} e_{tsm} \rightsquigarrow i_{tsm} \Leftarrow \mathtt{Parser}(\mathtt{Type} \times \tau_{md}) \end{array}}{d; \mathbf{tytsm}(s, n, \kappa, \tau_{md}, e_{tsm}) \sim (\Psi, s[\mathbf{ty}(n, \kappa, \tau_{md}, i_{tsm})]; \Theta)} \text{ (D-tytsm)}$$

$$\frac{d \sim (\Psi; \Theta) \qquad \vdash^{\Psi}_{\Theta_0 \Theta} \theta \sim \Upsilon \qquad \vdash^{\Psi}_{\Theta_0 \Theta} \theta \sim_{\Upsilon} \Theta' \qquad \vdash^{\Psi}_{\Theta_0 \Theta} \theta \sim_{\Theta'} \Theta''}{d; \mathbf{mrectydecl}(\theta) \sim (\Psi; \Theta\Theta'')} \text{ (D-rec)}$$

We formalize mutually recursive types with the rule above, following functional languages like Ocaml: all mutually recursive types should be declared together with the keyword `and` separating them. The rule operates in three passes through the list. Type-level TSMs must declare how many types they must be mutually defined against. Each application of a type-level TSM still only generates a single type.

$\boxed{\vdash^{\Psi}_{\Theta} \theta \sim \Upsilon}$ $\quad \Upsilon ::= \emptyset \mid \Upsilon, \mathtt{T} :: \kappa$

First the names and kinds are extracted and placed in a list, Υ.

$$\frac{}{\vdash^{\Psi}_{\Theta} \emptyset \sim \emptyset}$$

$$\frac{\vdash^{\Psi}_{\Theta} \theta \sim \Upsilon \qquad \mathtt{T} \notin \mathrm{dom}(\Upsilon) \qquad \mathtt{T} \notin \mathrm{dom}(\Theta) \qquad \emptyset \vdash^{\Psi}_{\Theta} \tau :: \kappa_1 \to \ldots \to \kappa_n \to \kappa}{\vdash^{\Psi}_{\Theta} \theta, \mathbf{tydecl}(\mathtt{T}, \tau, e) \sim \Upsilon, \mathtt{T} :: \kappa}$$

$$\frac{\vdash^{\Psi}_{\Theta} \theta \sim \Upsilon \qquad \mathtt{T} \notin \mathrm{dom}(\Upsilon) \qquad \mathtt{T} \notin \mathrm{dom}(\Theta) \qquad s[\mathbf{ty}(n, \kappa, \tau, i)] \in \Psi}{\vdash^{\Psi}_{\Theta} \theta, \mathbf{tyaptsm}(\mathtt{T}, s, \mathit{body}, e) \sim \Upsilon, \mathtt{T} :: \kappa}$$

$\boxed{\vdash^{\Psi}_{\Theta} \theta \sim_{\Upsilon} \Theta}$

Then, the types themselves are constructed and the type-level TSMs are applied. The metadata is not generated yet (though we save the metadata generated by a type-level TSM so we don't have to re-run it in the next phase.) Each type-level term must have a kind that takes in references to all the others.

$$\frac{}{\vdash^{\Psi}_{\Theta} \emptyset \sim_{\Upsilon} \emptyset}$$

$$\frac{\vdash^{\Psi}_{\Theta} \theta \sim_{\Upsilon} \Theta' \qquad \Upsilon = \mathtt{T}_1 :: \kappa_1, \ldots, \mathtt{T}_i :: \kappa_i, \ldots, \mathtt{T}_n :: \kappa_n \qquad \emptyset \vdash_{\Theta} \tau :: \kappa_1 \to \ldots \to \kappa_n \to \kappa_i}{\vdash^{\Psi}_{\Theta} \theta, \mathbf{tydecl}(\mathtt{T}_i, \tau, e) \sim_{\Upsilon} \Theta', \mathtt{T_i}[\tau(\mathtt{T}_1) \ldots (\mathtt{T}_n) :: \kappa_i, -]}$$

$$\frac{\vdash^{\Psi}_{\Theta} \theta \sim_{\Upsilon} \Theta' \qquad \Upsilon = \mathtt{T}_1 :: \kappa_1, \ldots, \mathtt{T}_i :: \kappa_i, \ldots, \mathtt{T}_n :: \kappa_n \qquad s[\mathbf{ty}(n, \kappa, \tau_{md}, i_{tsm})] \in \Psi \quad \mathsf{parsestream}(body) = i_{ps} \qquad i_{tsm}.parse(i_{ps}) \Downarrow OK((i_{type}, i_{md})) \qquad i_{type} \uparrow \hat{\tau} \qquad \emptyset; \emptyset \vdash_{\Theta_0 \Theta \Theta'} \hat{\tau} \rightsquigarrow \tau :: \kappa_1 \to \ldots \to \kappa_n \to \kappa_i}{\vdash^{\Psi}_{\Theta} \theta, \mathbf{tyaptsm}(\mathtt{T}_i, s, body, e) \sim_{\Upsilon} \Theta', \mathtt{T}_i[\tau(\mathtt{T}_i) \ldots (\mathtt{T}_n) :: \kappa_i, i_{md} : \tau_{md}]}$$

$\boxed{\vdash^{\Psi}_{\Theta} \theta \sim_{\Theta} \Theta}$

Finally, the metadata and metadata transformations are processed.

$$\frac{}{\vdash^{\Psi}_{\Theta} \emptyset \sim_{\Theta'} \emptyset}$$

$$\frac{\vdash^{\Psi}_{\Theta} \theta \sim_{\Theta', \mathtt{T}[\tau :: \kappa, -]} \Theta'' \qquad \emptyset; \emptyset \vdash^{\Psi}_{\Theta \Theta', \mathtt{T}[\tau :: \kappa, -]} e_{md} \rightsquigarrow i_{md} \Rightarrow \tau_{md}}{\vdash^{\Psi}_{\Theta} \theta, \mathbf{tydecl}(\mathtt{T}, \tau, e_{md}) \sim_{\Theta'', \mathtt{T}[\tau :: \kappa, -]} \Theta', \mathtt{T}[\tau :: \kappa, i_{md} : \tau_{md}]}$$

$$\frac{\vdash^{\Psi}_{\Theta} \theta \sim_{\Theta', \mathtt{T}[\tau :: \kappa, i_{md} : \tau_{md}]} \Theta'' \qquad \emptyset; \emptyset \vdash^{\Psi}_{\Theta \Theta', \mathtt{T}[\tau :: \kappa, i_{md} : \tau_{md}]} e_{mdx} \rightsquigarrow i_{mdx} \Rightarrow \tau_{md} \to \tau'_{md} \qquad i_{mdx}(i_{md}) \Downarrow i'_{md}}{\vdash^{\Psi}_{\Theta} \theta, \mathbf{tyaptsm}(\mathtt{T}, s, body, e_{mdx}) \sim_{\Theta', \mathtt{T}[\tau :: \kappa, i_{md} : \tau_{md}]} \Theta'', \mathtt{T}[\tau :: \kappa_i, i'_{md} : \tau'_{md}]}$$

$\boxed{\Delta \vdash_{\Theta} \tau :: \kappa}$

$$\frac{\vdash_{\Theta} \omega}{\Delta \vdash_{\Theta} \mathbf{objtype}[\omega] :: \star} \qquad \frac{\vdash_{\Theta} \chi}{\Delta \vdash_{\Theta} \mathbf{casetype}[\chi] :: \star}$$

$$\frac{\Delta \vdash_{\Theta} \tau_1 :: \star \qquad \Delta \vdash_{\Theta} \tau_2 :: \star}{\Delta \vdash_{\Theta} \tau_1 \to \tau_2 :: \star} \qquad \frac{\mathtt{T}[\tau :: \kappa, i : \tau] \in \Theta}{\Delta \vdash_{\Theta} \mathtt{T} :: \kappa}$$

$$\frac{\Delta \vdash_{\Theta} \tau_1 :: \star \qquad \Delta \vdash_{\Theta} \tau_2 :: \star}{\Delta \vdash_{\Theta} \tau_1 \times \tau_2 :: \star} \qquad \frac{\Delta \vdash_{\Theta} \tau_1 :: \kappa_1 \to \kappa_2 \qquad \Delta \vdash_{\Theta} \tau_2 :: \kappa_1}{\Delta \vdash_{\Theta} \tau_1(\tau_2) :: \kappa_2}$$

$$\frac{t :: \kappa \in \Delta}{\Delta \vdash_{\Theta} t :: \kappa} \qquad \frac{\Delta, t : \kappa_1 \vdash_{\Theta} \tau :: \kappa_1}{\Delta \vdash_{\Theta} \lambda[\kappa_1](t.\tau) :: \kappa \to \kappa_2} \qquad \frac{\Delta, t :: \star \vdash_{\Theta} \tau :: \star}{\Delta \vdash_{\Theta} \forall(t.\tau) :: \star}$$

$\boxed{\Delta; \Delta \vdash_{\Theta} \hat{\tau} \rightsquigarrow \tau :: \kappa}$

$$\frac{\Delta_{out}; \Delta \vdash_{\Theta} \hat{\omega} \rightsquigarrow \omega}{\Delta_{out}; \Delta \vdash_{\Theta} \mathbf{objtype}[\hat{\omega}] \rightsquigarrow \mathbf{objtype}[\omega] :: \star}$$

$$\frac{\Delta_{out}; \Delta \vdash_{\Theta} \hat{\chi} \rightsquigarrow \chi}{\Delta_{out}; \Delta \vdash_{\Theta} \mathbf{casetype}[\hat{\chi}] \rightsquigarrow \mathbf{casetype}[\chi] :: \star}$$

$$\frac{\Delta_{out}; \Delta \vdash_{\Theta} \hat{\tau}_1 \rightsquigarrow \tau_1 :: \star \qquad \Delta_{out}; \Delta \vdash_{\Theta} \hat{\tau}_2 \rightsquigarrow \tau_2 :: \star}{\Delta_{out}; \Delta \vdash_{\Theta} \hat{\tau}_1 \to \hat{\tau}_2 \rightsquigarrow \tau_1 \to \tau_2 :: \star} \qquad \frac{\mathtt{T}[\tau :: \kappa, i : \tau] \in \Theta}{\Delta_{out}; \Delta \vdash_{\Theta} \mathtt{T} \rightsquigarrow \mathtt{T} :: \kappa}$$

$$\frac{\Delta_{out}; \Delta \vdash_{\Theta} \hat{\tau}_1 \rightsquigarrow \tau_1 :: \star \qquad \Delta_{out}; \Delta \vdash_{\Theta} \hat{\tau}_2 \rightsquigarrow \tau_2 :: \star}{\Delta_{out}; \Delta \vdash_{\Theta} \hat{\tau} \times \hat{\tau} \rightsquigarrow \tau_1 \times \tau_2 :: \star}$$

$$\frac{\Delta_{out}; \Delta \vdash_{\Theta} \hat{\tau}_1 \rightsquigarrow \tau_1 :: \kappa_1 \to \kappa_2 \qquad \Delta_{out}; \Delta \vdash_{\Theta} \hat{\tau}_2 \rightsquigarrow \tau_2 :: \kappa_1}{\Delta_{out}; \Delta \vdash_{\Theta} \hat{\tau}_1(\hat{\tau}_2) \rightsquigarrow \tau_1(\tau_2) :: \kappa_2}$$

$$\frac{t :: \kappa \in \Delta}{\Delta_{out}; \Delta \vdash_{\Theta} t :: \kappa} \qquad \frac{\Delta_{out}; \Delta, t : \kappa_1 \vdash_{\Theta} \hat{\tau} \rightsquigarrow \tau :: \kappa_1}{\Delta_{out}; \Delta \vdash_{\Theta} \lambda[\kappa_1](t.\hat{\tau}) \rightsquigarrow \lambda[\kappa_1](t.\tau) :: \kappa \to \kappa_2}$$

$$\frac{\Delta_{out}; \Delta, t :: \star \vdash_{\Theta} \hat{\tau} \rightsquigarrow \tau :: \star}{\Delta_{out}; \Delta \vdash_{\Theta} \forall(t.\hat{\tau}) \rightsquigarrow \forall(t.\tau) :: \star}$$

$$\frac{\Delta_{out} \vdash_{\Theta} \tau :: \kappa}{\Delta_{out}; \Delta \vdash_{\Theta} \mathbf{spliced}[\tau] \rightsquigarrow \tau :: \kappa}$$

$\boxed{i \uparrow \tau}$ $\boxed{\tau \downarrow i}$

$$\frac{i \uparrow \mathtt{T}}{\mathbf{iinj}[Named](i) \uparrow \mathtt{T}} \qquad \frac{\mathtt{T} \downarrow i_{id}}{\mathtt{T} \downarrow \mathbf{iinj}[Named](i_{id})}$$

$$\frac{i \uparrow \omega}{\mathbf{iinj}[Objtype](i) \uparrow \mathbf{objtype}[\omega]} \qquad \frac{\omega \downarrow i}{\mathbf{objtype}[\omega] \downarrow \mathbf{iinj}[Objtype](i)}$$

$$\frac{i \uparrow \chi}{\mathbf{iinj}[Casetype](i) \uparrow \mathbf{casetype}[\chi]} \qquad \frac{\chi \downarrow i}{\mathbf{casetype}[\omega] \downarrow \mathbf{iinj}[Casetype](i)}$$

$$\frac{i_1 \uparrow \tau_1 \qquad i_2 \uparrow \tau_2}{\mathbf{iinj}[Arrow]((i_1, i_2)) \uparrow \tau_1 \to \tau_2} \qquad \frac{\tau_1 \downarrow i_1 \qquad \tau_2 \downarrow i_2}{\tau_1 \to \tau_2 \downarrow \mathbf{iinj}[Arrow]((i_1, i_2))}$$

$$\frac{}{\mathbf{iinj}[Nil](i) \uparrow \emptyset} \qquad \frac{}{\emptyset \downarrow \mathbf{iinj}[Nil](i)}$$

$$\frac{\begin{array}{c} i_l \uparrow l \\ i_t \uparrow \tau \qquad i_c \uparrow \omega \end{array}}{\mathbf{iinj}[Cons]((i_l, i_t, i_c)) \uparrow l[\tau], \omega} \qquad \frac{l \downarrow i_l \qquad \tau \downarrow i_t \qquad \omega \downarrow i_c}{l[\tau], \omega \downarrow \mathbf{iinj}[Cons]((i_l, i_t, i_c))}$$

$$\frac{}{\mathbf{iinj}[Nil](i) \uparrow \emptyset} \qquad \frac{}{\emptyset \downarrow \mathbf{iinj}[Nil](i)}$$

$$\frac{\begin{array}{c} i_l \uparrow C \\ i_t \uparrow \tau \qquad i_c \uparrow \chi \end{array}}{\mathbf{iinj}[Cons]((i_l, i_t, i_c)) \uparrow C[\tau], \chi} \qquad \frac{C \downarrow i_l \qquad \tau \downarrow i_t \qquad \chi \downarrow i_c}{C[\tau], \chi \downarrow \mathbf{iinj}[Cons]((i_l, i_t, i_c))}$$

... ...

$$\frac{\mathsf{body}(i_{ps}) = body \qquad \mathsf{tparse}(body) = \tau}{\mathbf{iinj}[Spliced](i_{ps}) \uparrow \mathbf{spliced}[\tau]}$$

$\boxed{i \uparrow \hat{e}}$ $\boxed{i \downarrow i}$

$$\frac{i_{id} \uparrow x}{\mathbf{iinj}[Var](i_{id}) \uparrow x} \qquad \frac{x \downarrow i_{id}}{x \downarrow \mathbf{iinj}[Var](i_{id})}$$

$$\frac{i_1 \uparrow \tau \qquad i_2 \uparrow \hat{e}}{\mathbf{iinj}[Asc]((i_1, i_2)) \uparrow \mathbf{hasc}[\tau](\hat{e})} \qquad \frac{\tau \downarrow i_1 \qquad i \downarrow i_2}{\mathbf{iasc}[\tau](i) \downarrow \mathbf{iinj}Asc((i_1, i_2))}$$

$$\frac{i_{id} \uparrow x \qquad i \uparrow \hat{e}}{\mathbf{iinj}[Lam]((i_{id}, i)) \uparrow \mathbf{hlam}(x.\hat{e})} \qquad \frac{x \downarrow i_{id} \qquad i \downarrow i'}{\mathbf{ilam}(x.i) \downarrow \mathbf{iinj}[Lam]((i_{id}, i'))}$$

$$\frac{i_1 \uparrow \hat{e}_1 \qquad i_2 \uparrow \hat{e}_2}{\mathbf{iinj}[Ap]((i_1, i_2)) \uparrow \mathbf{hap}(\hat{e}_1, \hat{e}_2)} \qquad \frac{i_1 \downarrow i_1' \qquad i_2 \downarrow i_2'}{\mathbf{iap}(i_1; i_2) \downarrow \mathbf{iinj}[Ap]((i_1', i_2'))}$$

... ...

$$\frac{\mathsf{body}(i_{ps}) = body \qquad \mathsf{eparse}(body) = e}{\mathbf{iinj}[Spliced](i_{ps}) \uparrow \mathbf{spliced}[e]}$$

A.4 Context Formation

$\boxed{\vdash_\Theta \Psi}$

$$\frac{}{\vdash_\Theta \emptyset} \qquad \frac{\Theta_0 \subset \Theta \qquad \vdash_\Theta \Psi \qquad \nexists\, s.(s = s_0 \wedge s[\mathbf{ty}(\_,\_,\_)] \in \Psi) \qquad \vdash_\Theta \tau_{md} :: \star \qquad \emptyset;\emptyset \vdash_\Theta i \Leftarrow \texttt{Parser}(\texttt{Type} \times \tau_{md})}{\vdash_\Theta \Psi; s_0[\mathbf{ty}(n, \kappa, \tau_{md}, i_{tsm})]}$$

$$\frac{\Theta_0 \subset \Theta \qquad \vdash_\Theta \Psi \qquad \nexists\, s.(s = s_0 \wedge (s[\mathbf{syn}(\_,\_)] \in \Psi \vee s[\mathbf{ana}(\_)]) \in \Psi) \qquad \emptyset;\emptyset \vdash_\Theta i \Leftarrow \texttt{Parser}(\texttt{Exp})}{\vdash_\Theta \Psi; s_0[\mathbf{ana}(i)]}$$

$$\frac{\Theta_0 \subset \Theta \qquad \vdash_\Theta \Psi \qquad \nexists\, s.(s = s_0 \wedge (s[\mathbf{syn}(\_,\_)] \in \Psi \vee s[\mathbf{ana}(\_)]) \in \Psi) \qquad \emptyset \vdash_\Theta \tau :: \star \qquad \emptyset;\emptyset \vdash_\Theta i \Leftarrow \texttt{Parser}(\texttt{Exp})}{\vdash_\Theta \Psi; s_0[\mathbf{syn}(\tau, i)]}$$

$\boxed{\vdash_\Theta \Theta}$

$$\frac{}{\vdash_\Theta \emptyset} \qquad \frac{\vdash_\Theta \Theta' \qquad \mathtt{T} \notin \mathrm{dom}(\Theta\Theta') \qquad \emptyset \vdash_{\Theta'} \tau :: \kappa \qquad \emptyset;\emptyset \vdash_\Theta i_{md} \Leftarrow \tau_{md}}{\vdash_\Theta \Theta', \mathtt{T}[\tau :: \kappa, i_{md} : \tau_{md}]}$$

$\boxed{\vdash_\Theta \omega}$

$$\frac{}{\vdash_\Theta \emptyset} \qquad \frac{l \notin \mathrm{dom}(\omega) \qquad \emptyset \vdash_\Theta \tau :: \star \qquad \vdash_\Theta \omega}{\vdash_\Theta l[\mathbf{def}, \tau]; \omega} \qquad \frac{l \notin \mathrm{dom}(\omega) \qquad \emptyset \vdash_\Theta \tau :: \star \qquad \vdash_\Theta \omega}{\vdash_\Theta l[\mathbf{val}, \tau]; \omega}$$

$\boxed{\vdash_\Theta \chi}$

$$\frac{}{\vdash_\Theta \emptyset} \qquad \frac{C \notin \mathrm{dom}(\chi) \qquad \emptyset \vdash_\Theta \tau :: \star \qquad \vdash_\Theta \chi}{\vdash_\Theta C[\tau]; \chi}$$

$\boxed{\vdash_\Theta \Gamma}$

$$\frac{}{\vdash_\Theta \emptyset} \qquad \frac{\vdash_\Theta \Gamma \qquad \emptyset \vdash_\Theta \tau :: \star}{\vdash_\Theta \Gamma, x : \tau}$$

A.5 Statics for external terms

Statics for core lambda calculus can be referred to in [8]. And here we present elaboration rules for TSM and TSL extensions on Wyvern language: including the rule for TSL literals elaboration $\mathbf{lit}[body]$, rules for TSM parser access $\mathbf{etsmdef}[s]$, and TSM application $\mathbf{eaptsm}[s, body]$.

$$\frac{\Theta_0 \subset \Theta \qquad T[\tau :: \kappa, i_m : \texttt{HasTSL}] \in \Theta \qquad \mathsf{parsestream}(body) = i_{ps} \qquad i_m.parser.parse(i_{ps}) \Downarrow OK((i_{ast}, i'_{ps})) \qquad i_{ast} \uparrow \hat{e} \qquad \Delta;\emptyset;\Gamma;\emptyset \vdash_\Theta \hat{e} \rightsquigarrow i \Leftarrow \mathtt{T}}{\Delta;\Gamma \vdash_\Theta \mathbf{lit}[body] \rightsquigarrow i \Leftarrow \mathtt{T}} \text{ (T-lit)}$$

$$\frac{\Theta_0 \subset \Theta \qquad s[\mathbf{syn}(\tau, i)] \in \Psi}{\Delta;\Gamma \vdash^\Psi_\Theta \mathbf{etsmdef}[s] \rightsquigarrow i \Rightarrow \texttt{Parser}(\texttt{Exp})} \text{ (T-syntsmdef)}$$

$$\frac{\Theta_0 \subset \Theta \qquad s[\mathbf{ana}(i)] \in \Psi}{\Delta;\Gamma \vdash^\Psi_\Theta \mathbf{etsmdef}[s] \rightsquigarrow i \Rightarrow \texttt{Parser}(\texttt{Exp})} \text{ (T-anatsmdef)}$$

$$\frac{\Theta_0 \subset \Theta \qquad s[\mathbf{ty}(\kappa, \tau, i)] \in \Psi}{\Delta;\Gamma \vdash^\Psi_\Theta \mathbf{etsmdef}[s] \rightsquigarrow i \Rightarrow \texttt{Parser}(\texttt{Type})} \text{ (T-typetsmdef)}$$

$$\frac{\Theta_0 \subset \Theta \qquad s[\mathbf{ana}(i_{tsm})] \in \Theta \qquad \mathsf{parsestream}(body) = i_{ps} \qquad i_{tsm}.parse(i_{ps});\Downarrow OK((i_{ast}, i'_{ps})) \qquad i_{ast} \uparrow \hat{e} \qquad \Delta;\emptyset;\Gamma;\emptyset \vdash^\Psi_\Theta \hat{e} \rightsquigarrow i \Leftarrow \tau}{\Delta;\Gamma \vdash^\Psi_\Theta \mathbf{eaptsm}[s, body] \rightsquigarrow i \Leftarrow \tau} \text{ (T-ana)}$$

$$\frac{\Theta_0 \subset \Theta \qquad s[\mathbf{syn}(\tau, i_{tsm})] \in \Theta \qquad \mathsf{parsestream}(body) = i_{ps} \qquad i_{tsm}.parse(i_{ps}) \Downarrow OK((i_{ast}, i'_{ps})) \qquad i_{ast} \uparrow \hat{e} \qquad \Delta;\emptyset;\Gamma;\emptyset \vdash^\Psi_\Theta \hat{e} \rightsquigarrow i \Leftarrow \tau}{\Delta;\Gamma \vdash^\Psi_\Theta \mathbf{eaptsm}[s, body] \rightsquigarrow i \Rightarrow \tau} \text{ (T-syn)}$$

B. METATHEORY

THEOREM 2 (INTERNAL TYPE SAFETY). *If* $\vdash \Theta$, *and* $\emptyset;\emptyset \vdash_\Theta i \Leftarrow \tau$ *or* $\emptyset;\emptyset \vdash_\Theta i \Rightarrow \tau$, *then either* i `val` *or* $i \mapsto i'$ *such that* $\emptyset;\emptyset \vdash_\Theta i' \Leftarrow \tau$.

PROOF. The dynamics are standard, so the proof is by a standard preservation and progress argument. One exception is the proof for the term reification expression $\mathbf{etoast}(e)$:

- case $\mathbf{etoast}[e]$: If $\Theta_0 \subset \Theta$ and $\emptyset;\emptyset \vdash_\Theta i \Leftarrow \tau$ then $i \downarrow i'$ and $\emptyset \vdash_\Theta i' \Leftarrow \texttt{Exp}$.
  The proof of the case can be referred to the rules $i \downarrow i$: for each interanl expression, there exists a rule to transform the expression into an AST presentation. And by induction on the derivation of the terms, the proof can be easily achieved.

□

THEOREM 3 (EXTERNAL TYPE PRESERVATION). *If* $\vdash \Theta$, $\vdash_{\Theta_0\Theta} \Psi$, $\vdash_{\Theta_0\Theta} \Gamma$, $\vdash \Delta$ *and* $\Delta;\Gamma \vdash^{\Psi}_{\Theta_0\Theta} e \rightsquigarrow i \Leftarrow \tau$ *or* $\Delta;\Gamma \vdash^{\Psi}_{\Theta_0\Theta} e \rightsquigarrow i \Rightarrow \tau$ *then* $\Delta;\Gamma \vdash_{\Theta_0\Theta} i \Leftarrow \tau$.

PROOF. Base on the proof of core Wyvern external terms in TSL, we only need to present the proofs for the new cases extended in external Wyvern:

- $\mathbf{eaptsm}[s, body]$, in this case, according to the rule T-syn, there exists a translational term $\hat{e}$, s.t. $\Delta;\emptyset;\Gamma;\emptyset \vdash^{\Psi}_{\Theta} \hat{e} \rightsquigarrow i \Leftarrow \tau$. For the typing context, we have $\vdash \emptyset$ (which is Γ_{out}) and $\vdash \emptyset$ (empty Δ_{out}). And by the conditions in the theorem $\vdash_\Theta \Gamma$, $\vdash \Theta$ and $\vdash_{\Theta_0\Theta} \Psi$, by Lemma 1, we have $\emptyset;\Gamma \vdash_{\Theta_0\Theta} i \Leftarrow \tau$.
- $\mathbf{etsmdef}[s]$. There are three subcases depends on the property of s: type-level TSM, synthetic TSM or analytic TSM. For synthetic TSM, by induction, we have the formation of the internal term i in $s[\mathbf{syn}(\tau, i)]$, i.e. $\vdash_{\Theta_0\Theta} i \Leftarrow \texttt{ExpParser}$ as $\Theta_0\Theta$ is well formed. The subcase for analytic TSM and type level TSM are similar, as the formation of the term i in $s[\mathbf{ana}(i)]$ and the term i in $s[\mathbf{ty}(\kappa, \tau, i)]$ are checked in declarations elaboration. Thus the case is proved.

With all these cases proved, we have the property holds for all external terms. □

LEMMA 1 (TRANSLATIONAL TYPE PRESERVATION). *If* $\vdash \Theta$, $\vdash_{\Theta_0\Theta} \Psi$, $\vdash_{\Theta_0\Theta} \Gamma_{out}$, $\vdash_{\Theta_0\Theta} \Gamma$, $dom(\Gamma_{out}) \cap dom(\Gamma) = \emptyset$, $\vdash \Delta$, $\vdash \Delta_{out}$, $dom(\Delta) \cap dom(\Delta_{out}) = \emptyset$ *and* $\Delta_{out};\Delta;\Gamma_{out};\Gamma \vdash^{\Psi}_{\Theta\Theta_0} \hat{e} \rightsquigarrow i \Leftarrow \tau$ *or* $\Delta_{out};\Delta;\Gamma_{out};\Gamma \vdash^{\Psi}_{\Theta\Theta_0} \hat{e} \rightsquigarrow i \Rightarrow \tau$ *then* $(\Delta_{out}\Delta);(\Gamma_{out}\Gamma) \vdash_{\Theta_0\Theta} i \Leftarrow \tau$.

PROOF. The proof by induction over the typing derivation for all the shared cases. The outer context is threaded through opaquely when applying the inductive hypothesis. By induction on all translational terms, we can easily prove them based on their derivation rules.

The only rules of note are the rules for the spliced external terms, which require applying the external type preservation theorem recursively. This is well-founded by a metric measuring the size of the spliced external term, written in concrete syntax, since we know it was derived from a portion of the literal body. □

LEMMA 2 (TRANSLATIONAL TYPE ELABORATION). *If* $\vdash \Theta$, $\vdash \Delta$, $\vdash \Delta_{out}$, $dom(\Delta) \cap dom(\Delta_{out}) = \emptyset$ *and* $\Delta_{out};\Delta \vdash_{\Theta_0\Theta} \hat{\tau} \rightsquigarrow \tau :: \kappa$, *then we have* $\Delta_{out}\Delta \vdash_{\Theta_0\Theta} \tau :: \kappa$.

PROOF. By induction on the derivation of the translational types, we can easily proof the cases in by checking the properties on the derivation rules:

- For the type $\mathbf{spliced}[\tau]$, the well-formedness is checked under the context Δ_{out}, i.e. $\Delta_{out}\tau$, thus in the check rule for internal types, we have $\Delta_{out}\Delta \vdash_\Theta \tau$. And the case is proved.
- For other terms, by induction, the proof is standard, as they are checked in Δ. For example, the formation of the arrow type $\tau_1 \to \tau_2$ is proved by its subterms τ_1 and τ_2. Thus the case is proved.

With these cases proved, the lemma is proved. □

THEOREM 4 (COMPILATION). *If* $\rho \sim (\Psi;\Theta) \rightsquigarrow i : \tau$ *then* $\vdash \Theta$, $\vdash_{\Theta_0\Theta} \Psi$ *and* $\emptyset;\emptyset \vdash_{\Theta_0\Theta} i \Leftarrow \tau$.

PROOF. The proof contains two parts: the formation of the contexts (Θ, Ψ) and the formation of the term i.

The proof for the contexts formation requires the following lemma. And the proof for the term i is obvious: According to the rule 'compile', we have that $\emptyset;\emptyset \vdash^{\Psi}_{\Theta_0\Theta} e \rightsquigarrow i \Rightarrow \tau$, and for the context: $\vdash \emptyset$ by the checking rule for Γ, $\vdash \emptyset$ for context Δ, $\vdash_{\Theta_0\Theta} \Psi$, $\vdash \Theta_0\Theta$ by the proof of the first part. Then by Theorem 1, we have $\emptyset;\emptyset \vdash_{\Theta_0\Theta} i \Leftarrow \tau$, which completes the proof. □

LEMMA 3. *If* $d \sim (\Psi';\Theta')$, *then we have* $\vdash \Theta'$ *and* $\vdash_{\Theta_0\Theta'} \Psi'$.

PROOF. By induction on the length of the derivation steps the declarations d. Check the last step of derivation, we have the following four cases to proof:

- $d = d_1; \mathbf{syntsm}(s, \tau, e_{tsm})$. To prove the well-formedness of the declaration, we need to prove that no name conflicts exists and the parser i is of type $\texttt{Parser}(\texttt{Exp})$.
  Firstly, by induction, we have the well-formedness of the previous contexts: $d_1 \sim (\Theta_1;\Psi_1)$ imply that Θ_1 and Ψ_1 is well formed. Secondly, in the rule D-syntsm, we check that its name does not appear in the previous context Ψ_1, thus we have $s[\mathbf{syn}(\_,\_)] \notin \text{dom}(\Psi_1) \vee s[\mathbf{ana}(\_)] \notin \text{dom}(\Psi_1)$, thirdly, by theorem 2 and the premise $\emptyset;\emptyset \vdash^{\Psi}_{\Theta_0\Theta} e_{tsm} \rightsquigarrow i_{tsm} \Leftarrow \texttt{Parser}(\texttt{Exp})$, we have $\emptyset;\emptyset \vdash_\Theta i \Leftarrow \texttt{Parser}(\texttt{Exp})$. And these three rules prove the well-formedness of the environment $\Psi_1, s[\mathbf{syn}(\tau, i)]$. Also by induction, we have $\vdash \Theta_1$. Combining these two conditions, the case is proved.

- $d = d_1; \mathbf{anatsm}(s, e_{tsm})$, this case is exactly the same with the previous case, expect that the type τ is omitted.
- $d = d_1; \mathbf{tytsm}(s, \kappa, \tau, e)$. To check the well-formedness of the type-level TSM, we need to check 1) No name conflicts exists in the previous context, 2) the type should be of kind $\star$, 3) the type of the parser i should be $\texttt{Parser}(\texttt{Type} \times \tau)$.

  The proof can be done with the following conditions in the derivation rules (D-tytsm): 1) The name is checked to be free of conflicts in the previous context. 2) The type τ is checked to be well formed and of kind $\star$. 3) By Theorem 2, and the premise $\emptyset; \emptyset \vdash_{\Theta_0 \Theta} e_{tsm} \rightsquigarrow i_{tsm} \Leftarrow \texttt{Parser}(\texttt{Type} \times \tau)$, we have $\emptyset; \emptyset \vdash_{\Theta_0 \Theta} i_{tsm} \Leftarrow \texttt{Parser}(\texttt{Type} \times \tau)$. And by induction, we have the formation of the previous context Θ_1 and Ψ_1. Thus the case is proved.
- $d = d; \mathbf{mrectydecl}(\theta_1, ..., \theta_n)$. The formation of a type declaration in Θ includes 1) no name conflicts in the previous context, 2) the type structure is well formed, 3) the metadata expression is well formed.

  Firstly, the names are check by the premise $\nexists\, u, v \in \{1, ..., n\}.(u \neq v \wedge \texttt{T}_u = \texttt{T}_v)$ and $\forall\, k.(\texttt{T}_k \notin \text{dom}(\Theta))$. Thus no name conflicts exists in the previous context.

  Secondly, by the rule (D-tydecl-1), we have $\emptyset \vdash_{\Theta\Theta_0} \tau :: \kappa$. By the rule (D-aptsm-1), we have $\emptyset \vdash_{\Theta\Theta_0} \hat{\tau} \rightsquigarrow \tau :: \kappa$, then by Lemma 2, we have $\emptyset \vdash_{\Theta\Theta_0} \tau :: \kappa$. Then by the type checking condition $\emptyset \vdash \mathsf{type}(\mathring{\theta}_k) :: \kappa_1 \to ... \to \kappa_n \to \kappa_k$, all types are of kind $\tau_1 \to ... \to \tau_n \to \tau_k$ and are well formed. Then by the named type substitution, we have the formation of the type $\tau_k(\texttt{T}_1, ..., T_n) :: \kappa_k$, which proves the well-formedness of each type structure.

  Thirdly, by the rule (D-tydecl-2), the metadata is checked by the condition $\emptyset; \emptyset \vdash^{\Psi}_{\Theta} e \rightsquigarrow i \Rightarrow \tau$, then by Theorem 2, we have $\emptyset; \emptyset \vdash_{\Theta} i \Leftarrow \tau$. And by (D-aptsm-2), the metadata i is well formed by applying a well formed term i_{mdx} to another well formed term i_{md}, thus the result should be well formed.

  Then by induction, we have the TSM context Ψ_1 untouched and thus it is well formed.

  With these three conditions proved, the case is proved.

With these four cases proved, the lemma is proved: compilation always produce well formed contexts. □