# Safely Composable Type-Specific Languages

Cyrus Omar, Darya Kurilova, Ligia Nistor, Benjamin Chung, and
Alex Potanin,[1] and Jonathan Aldrich

Carnegie Mellon University and Victoria University of Wellington[1]
{comar, darya, lnistor, bwchung, aldrich}@cs.cmu.edu and alex@ecs.vuw.ac.nz[1]

**Abstract.** Programming languages often include specialized notation for common datatypes (e.g. lists) and some also build in support for specific specialized datatypes (e.g. regular expressions), but user-defined types must use general-purpose notations. Frustration with this causes developers to use strings, rather than structured representations, with alarming frequency, leading to correctness, performance, security, and usability issues. Allowing library providers to modularly extend a language with new notations could help address these issues. Unfortunately, prior mechanisms either limit expressiveness or are not safely composable: individually unambiguous extensions can still lead to ambiguities when used together. We introduce *type-specific languages* (TSLs): logic associated with a type that determines how the bodies of *generic literals*, able to contain arbitrary syntax, are parsed and elaborated, hygienically. The TSL for a type is invoked only when a literal appears where a term of that type is expected, guaranteeing non-interference. We give evidence supporting the applicability of this approach and formally specify it with a bidirectionally typed elaboration semantics for the Wyvern language.

**Keywords:** extensible languages; parsing; bidirectional typechecking

## 1 Motivation

Many data types can be seen, semantically, as modes of use of general purpose product and sum types. For example, lists can be seen as recursive sums by observing that a list can either be empty, or be broken down into a product of the *head* element and the *tail*, another list. In an ML-like functional language, sums are exposed as datatypes and products as tuples and records, so list types can be defined as follows:

```
datatype 'a list = Nil | Cons of 'a * 'a list
```

In class-based object-oriented language, objects can be seen as products of their instance data and classes as the cases of a sum type [9]. In low-level languages, like C, structs and unions expose products and sums, respectively.

By defining user-defined types in terms of these general purpose constructs, we immediately benefit from powerful reasoning principles (e.g. induction), language support (e.g. pattern matching) and compiler optimizations. But these semantic benefits often come at a syntactic cost. For example, few would claim that writing a list of numbers as a sequence of `Cons` cells is convenient:

```
Cons(1, Cons(2, Cons(3, Cons(4, Nil))))
```

Lists are a common data structure, so many languages include *literal syntax* for introducing them, e.g. `[1, 2, 3, 4]`. This syntax is semantically equivalent to the general-purpose syntax shown above, but brings cognitive benefits both when writing and reading code by focusing on the content of the list, rather than the nature of the encoding. Using terminology from Green's cognitive dimensions of notations [8], it is more *terse*, *visible* and *maps more closely* to the intuitive notion of a list. Stoy, in discussing the value of good notation, writes [30]:

> A good notation thus conceals much of the inner workings behind suitable abbreviations, while allowing us to consider it in more detail if we require: matrix and tensor notations provide further good examples of this. It may be summed up in the saying: "A notation is important for what it leaves out."

Although list, number and string literals are nearly ubiquitous features of modern languages, some languages provide specialized literal syntax for other common collections (like maps, sets, vectors and matrices), external data formats (like XML and JSON), query languages (like regular expressions and SQL), markup languages (like HTML and Markdown) and many other types of data. For example, a language with built-in notation for HTML and SQL, supporting type safe *splicing* via curly braces, might define:

```
1  let webpage : HTML = <html><body><h1>Results for {keyword}</h1>
2    <ul id="results">{to_list_items(query(db,
3      SELECT title, snippet FROM products WHERE {keyword} in title))}
4    </ul></body></html>
```

as shorthand for:

```
1  let webpage : HTML = HTMLElement(Dict.empty(), [BodyElement(Dict.empty(),
2    [H1Element(Dict.empty(), [TextNode("Results for " + keyword)]),
3    ULElement((Dict.add Dict.empty() ("id","results")), to_list_items(query(db,
4      SelectStmt(["title", "snippet"], "products",
5        [WhereClause(InPredicate(StringLit(keyword), "title"))]))))])])
```

When general-purpose notation like this is too cognitively demanding for comfort, but a specialized notation as above is not available, developers turn to run-time mechanisms to make constructing data structures more convenient. Among the most common strategies in these situations, no matter the language paradigm, is to simply use a string representation, parsing it at run-time:

```
1  let webpage : HTML = parse_html("<html><body><h1>Results for "+keyword+"</h1>
2    <ul id=\"results\">" + to_string(to_list_items(query(db, parse_sql(
3      "SELECT title, snippet FROM products WHERE '"+keyword+"' in title")))) +
4    "</ul></body></html>")
```

Though recovering some of the notational convenience of the literal version, it is still more awkward to write, requiring explicit conversions to and from structured representations (`parse_html` and `to_string`, respectively) and escaping when the syntax of the data language interferes with the syntax of string literals (line 2). Such code also causes a number of problems that go beyond cognitive load. Because parsing occurs at run-time, syntax errors will not be discovered statically, causing potential run-time errors in production scenarios. Run-time parsing also incurs performance overhead, particularly relevant when code like this is executed often (as on a heavily-trafficked website). But the most serious issue with this code is that it is highly insecure: it is

vulnerable to cross-site scripting attacks (line 1) and SQL injection attacks (line 3). For example, if a user entered the keyword `'; DROP TABLE products --`, the entire product database could be erased. These attack vectors are considered to be two of the most serious security threats on the web today [25]. Although developers are cautioned to sanitize their input, it can be difficult to verify that this was done correctly throughout a codebase. The best way to avoid these problems today is to avoid strings and other similar conveniences and insist on structured representations. Unfortunately, situations like this, where maintaining strong correctness, performance and security guarantees entails significant syntactic overhead, causing developers to turn to less structured solutions that are more convenient, are quite common (as we will discuss in Sec. 5).

Adding new literal syntax into a language is generally considered to be the responsibility of the language's designers. This is largely for technical reasons: not all syntactic forms can unambiguously coexist in the same grammar, so a designer is needed to decide which syntactic forms are available, and what their semantics should be. For example, conventional notations for sets and maps are both delimited by curly braces. When Python introduced set literals, it chose to distinguish them based on whether the literal contained only values (e.g. `{3}`), or key-value pairs (`{"x": 3}`). But this causes an ambiguity with the syntactic form `{ }` – should it mean an empty set or an empty map (called a dictionary in Python)? The designers of Python avoided the ambiguity by choosing the latter interpretation (in this case, for backwards compatibility reasons).

Were this power given to library providers in a decentralized, unconstrained manner, the burden of resolving ambiguities would instead fall on developers who happened to import conflicting extensions. Indeed, this is precisely the situation with SugarJ [6] and other extensible languages generated by Sugar* [7], which allow library providers to extend the base syntax of the host language with new forms in a relatively unconstrained manner. These new forms are imported transitively throughout a program. To resolve syntactic ambiguities that arise, clients must manually augment the composed grammar with new rules that allow them to choose the correct interpretation explicitly. This is both difficult to do, requiring a reasonably thorough understanding of the underlying parser technology (in Sugar*, generalized LR parsing) and increases the cognitive load of using the conflicting notations (e.g. both sets and maps) together because disambiguation tokens must be used. These kinds of conflicts occur in a variety of circumstances: HTML and XML, different variants of SQL, JSON literals and maps, or differing implementations ("desugarings") of the same syntax (e.g. two regular expression engines). Code that uses these common abstractions together is very common in practice [13].

In this work, we will describe an alternative parsing strategy that sidesteps these problems by building into the language only a delimitation strategy, which ensures that ambiguities do not occur. The parsing and elaboration of literal bodies occurs during typechecking, rather than in the initial parsing phase. In particular, the typechecker defers responsibility to library providers, by treating the body of the literal as a term of the *type-specific language (TSL)* associated with the type it is being checked against. The TSL definition is responsible for elaborating this term using only general-purpose syntax. This strategy permits significant semantic flexibility – the meaning of a form like `{ }` can differ depending on its type, so it is safe to use it for empty sets, maps and

JSON literals. This frees these common forms from being tied to the variant of a data structure built into a language's standard library, which may not provide the precise semantics that a programmer needs (for example, Python dictionaries do not preserve key insertion order).

We present our work as a variant of an emerging programming language called Wyvern [22]. To allow us to focus on the essence of our proposal and provide the community with a minimal foundation for future work, the variant of Wyvern we develop here is simpler than the variant we previously described: it is purely functional (there are no effects other than non-termination) and it does not enforce a uniform access principle for objects (fields can be accessed directly), so objects are essentially just recursive labeled products with simple methods. It also adds recursive sum types, which we call *case types*, similar to those found in ML. One can refer to our version of the language as *TSL Wyvern* when the variant being discussed is not clear. Our work substantially extends and makes concrete a mechanism we sketched in a short workshop paper [23].

The paper is organized as a language design for TSL Wyvern:

- In Sec. 2, we introduce TSL Wyvern with a practical example. We introduce both inline and forward referenced literal forms, splicing, case and object types and an example of a TSL definition.
- In Sec. 3, we specify the layout-sensitive concrete syntax of TSL Wyvern with an Adams grammar and introduce the abstract syntax of TSL Wyvern.
- In Sec. 4, we specify the static semantics of TSL Wyvern as a *bidirectionally typed elaboration semantics*, which combines two key technical mechanisms:
  1. **Bidirectional Typechecking**: By distinguishing locations where an expression must synthesize a type from locations where an expression is being analyzed against a known type, we precisely specify where generic literals can appear and how dispatch to a TSL definition (an object with a parse method serving as metadata of a type) occurs.
  2. **Hygienic Elaboration**: Elaboration of literals must not cause the inadvertent capture or shadowing of variables in the context where the literal appears. It must, however, remain possible for the client to do so in those portions of the literal body treated as spliced expressions. The language cannot know *a priori* where these spliced portions will be. We give a clean type-theoretic formulation that achieves of this notion of hygiene.
- In Sec. 5, we gather initial data on how broadly applicable our technique may be by conducting a corpus analysis, finding that existing code often uses strings where specialized syntax might be more appropriate.
- In Sec. 6, we briefly report on the current implementation status of our work.
- We discuss related work in Sec. 7 and conclude in Sec. 8 with a discussion of present limitations and future research directions.

## 2 Type-Specific Languages in Wyvern

We begin with an example in Fig. 1 showing several different TSLs being used to define a fragment of a web application showing search results from a database. We will review this example below to develop intuitions about TSLs in Wyvern; a formal and more detailed description will follow in the subsequent sections. Note that for clarity of

```
1  let imageBase : URL = <images.example.com>
2  let bgImage : URL = <%imageBase%/background.png>
3  new : SearchServer
4    def resultsFor(searchQuery, page)
5      serve(~) (* serve : HTML -> Unit *)
6        >html
7          >head
8            >title Search Results
9            >style ~
10             body { background-image: url(%bgImage%) }
11             #search { background-color: %darken(‘#aabbcc‘, 10pct)% }
12         >body
13           >h1 Results for <{HTML.Text(searchQuery)}:
14           >div[id="search"]
15             Search again: < SearchBox("Go!")
16           < (* fmt_results : DB * SQLQuery * Nat * Nat -> HTML *)
17               fmt_results(db, ~, 10, page)
18                 SELECT * FROM products WHERE {searchQuery} in title
```

Fig. 1: Wyvern Example with Multiple TSLs

```
<literal body here, <inner angle brackets> must be balanced>
{literal body here, {inner braces} must be balanced}
[literal body here, [inner brackets] must be balanced]
‘literal body here, ‘‘inner backticks‘‘ must be doubled‘
’literal body here, ’’inner single quotes’’ must be doubled’
"literal body here, ""inner double quotes"" must be doubled"
12xyz (* no delimiters necessary for number literals; suffix optional *)
```

Fig. 2: Inline Generic Literal Forms

presentation, we color each character according to the TSL it is governed by. Black is the base language and comments are in italics.

### 2.1 Inline Literals

Our first TSL appears on the right-hand side of the variable binding on line 1. The variable `imageBase` is annotated with its type, `URL`. This is a named object type declaring several fields representing the components of a URL: its protocol, domain name, port, path and so on (not shown). We could have created a value of type `URL` using general-purpose notation (using the keyword **`new`**, an expression form that *forward references* an indented block of field and method definitions beginning on the line after it appears):

```
1  objtype URL
2    val protocol : String
3    val subdomain : String
4    (* ... *)
```

```
1  let imageBase = new : URL
2    val protocol = "http"
3    val subdomain = "images"
4    (* ... *)
```

This is tedious. By associating a TSL with the `URL` type (we will show how later), we can instead introduce precisely this value using conventional notation for URLs by placing it in the *body* of a *generic literal*, `<images.example.com>`. Any other delimited form in Fig. 2 can equivalently be used when the constraints shown can be obeyed. The type annotation on `imageBase` (or equivalently, directly after the literal) implies that this literal's *expected type* is `URL`, so the body of the literal (the characters between the angle brackets, in blue) will be governed by the `URL` TSL during the typechecking phase. This TSL will parse the body (at compile-time) to produce a Wyvern abstract syntax

tree (AST) that explicitly instantiates a new object of type `URL` using general-purpose notation as if the above had been written directly.

### 2.2 Splicing

In addition to supporting conventional notation for URLs, this TSL supports *splicing* another Wyvern expression of type `URL` to form a larger URL. The spliced term is delimited by percent signs, as seen on line 2 of Fig. 1. The TSL parses code between percent signs as a Wyvern expression, using its abstract syntax tree (AST) to construct an AST for the expression as a whole. A string-based representation of the URL is never used at run-time. Note that the delimiters used to go from Wyvern to a TSL are controlled by Wyvern while the TSL controls how to return to Wyvern.

### 2.3 Layout-Delimited Literals

On line 5 of Fig. 1, we see a call to a function `serve` (not shown) which has type `HTML -> Unit`. Here, `HTML` is a user-defined *case type*, having cases for each HTML tag as well as some other structures, such as text nodes and sequencing. Declarations of some of these cases can be seen on lines 2-6 of Fig. 4 (note that TSL Wyvern also includes simple product types for convenience, written `T1 * T2`). We could again use Wyvern's general-purpose introductory form for case types, e.g. `HTML.BodyElement((attrs, child))` (unlike in ML, in Wyvern we must explicitly qualify constructors with the case type they are part of when they are used. This is largely to make our formal semantics simpler and for clarity of presentation.) But, as discussed above, using this syntax can be inconvenient and cognitively demanding. Thus, we associate a TSL with `HTML` that provides a simplified notation for writing HTML, shown being used on lines 6-18 of Fig. 1. This literal body is layout-delimited, rather than delimited by explicit tokens as in Fig. 2, and introduced by a form of *forward reference*, written ~ ("tilde"), on the previous line. Because the forward reference occurs in a position where the expected type is `HTML`, the literal body is governed by that type's TSL. The forward reference will be replaced by the general-purpose term, of type `HTML`, generated by the TSL during typechecking. Because layout was used as a delimiter, there are no syntactic constraints on the body, unlike with inline forms (Fig. 2). For HTML, this is quite useful, as all of the inline forms impose constraints that would cause conflict with some valid HTML.

### 2.4 Implementing a TSL

Portions of the implementation of the TSL for `HTML` are shown on lines 8-15 of Fig. 4. A TSL is associated with a named type, forming an *active type*, using a more general mechanism for associating a pure, static value with a named type, called its *metadata*. Metadata is introduced as shown on line 8 of Fig. 4. Type metadata, in this context, is comparable to class annotations in Java or attributes in C#/F# and internalizes the practice of writing metadata using comments, so that it can be checked by the language and accessed programmatically more easily. This can be used for a variety of purposes – to associate documentation with a type, to mark types as being deprecated, and so on.

For the purposes of this work, metadata values will always be of type `HasTSL`, an object type that declares a single field, `parser`, of type `Parser`. The `Parser` type is an object type declaring a single method, `parse`, that transforms a `ParseStream` extracted

```
casetype HTML
  Empty
  Seq of HTML * HTML
  Text of String
  BodyElement of Attributes * HTML
  StyleElement of Attributes * CSS
  (* ... *)
  metadata = new : HasTSL
    val parser = ~
      start <- '>body'= attributes start>
        fn (attrs, child) => `BodyElement((%attrs%, %child%))`
      start <- '>style'= attributes EXP>
        fn (attrs, e) => `StyleElement((%attrs%, %e%))`
      start <- '<'= EXP>
        fn (e) => `%e% : HTML`
```

Fig. 4: A Wyvern case type with an associated TSL.

```
objtype HasTSL
  val parser : Parser
objtype Parser
  def parse(ps : ParseStream) : Result
  metadata : HasTSL = new
    val parser = (*parser generator*)
casetype Result
  OK of Exp * ParseStream
  Error of String * Location
```

```
casetype Exp
  Var of ID
  Lam of ID * Type * Exp
  Ap of Exp * Exp
  New of Members
  ...
  Spliced of ParseStream
  metadata : HasTSL = new
    val parser = (*quasiquotes*)
```

Fig. 5: Some of the types included in the Wyvern prelude.

from a literal body to a Wyvern AST. An AST is a value of type `Exp`, a case type that encodes the abstract syntax of Wyvern expressions. Fig. **??** shows portions of the declarations of these types, which live in the Wyvern *prelude* (a collection of types that are automatically loaded before any other).

Notice, however, that the TSL for `HTML` is not provided as an explicit `parse` method but instead as a declarative grammar. A grammar is a specialized notation for defining a parser, so we can implement a more convenient grammar-based parser generator as a TSL associated with the `Parser` type. We chose the layout-sensitive formalism developed by Adams [1] – Wyvern is itself layout-sensitive and has a grammar that can be written down using this formalism, so it is sensible to expose it to TSL providers as well. Most aspects of this formalism are completely conventional. Each non-terminal (e.g. `start`) is defined by a number of disjunctive productions, each introduced using `<-`. Each production defines a sequence of terminals (e.g. `'>body'`) and non-terminals (e.g. `start`, or one of the built-in non-terminals `ID`, `EXP` or `TYPE`, representing Wyvern identifiers, expressions and types, respectively). Unique to Adams grammars is that each terminal and non-terminal in a production can also have an optional *layout constraint* associated with it. The layout constraints available are `=` (meaning that the leftmost column of the annotated term must be aligned with that of the parent term), `>` (the leftmost column must be indented further) and `>=` (the leftmost column *may* be indented further). We will discuss this formalism further when we formally specify Wyvern's layout-sensitive concrete syntax.

Each production is followed, in an indented block, by a Wyvern function that generates a value given the values recursively generated by each of the $n$ non-terminals

it contains, ordered left-to-right. For the starting non-terminal, always called `start`, this function must return a value of type `Exp`. User-defined non-terminals might have a different type associated with them (not shown). Here, we show how to generate an AST using general-purpose notation for `Exp` (lines 13-15) as well as a more natural *quasiquote* style (lines 11 and 18). Quasiquotes are expressions that are not evaluated, but rather reified into syntax trees. We observe that quasiquotes too fall into the pattern of "specialized notation associated with a type" – quasiquotes for expressions, types and identifiers are simply TSLs associated with `Exp`, `Type` and `ID` (Fig. **??**). They support the full Wyvern concrete syntax as well as an additional delimited form, written with `%s`, that supports "unquoting": splicing another AST into the one being generated. Again, splicing is safe and structural, rather than based on string concatenation.

We have now seen several examples of TSLs that support splicing. The question then arises: what type should the spliced Wyvern expression be expected to have? This is determined by placing the spliced value in a place in the generated AST where its type is known – on line 11 of Fig. 4 it is known to be `HTML` and on line 13 it is known to be `CSS` by the declaration of `HTML`, and on line 15, it is known to be `HTML` by the use of an explicit ascription. When these generated ASTs are recursively typechecked during compilation, any use of a nested TSL at the top-level (e.g. the CSS TSL in Fig 1) will operate as intended.

### 2.5 Implementing Splicing

We have now seen several examples of splicing. Within the TSL for `HTML`, we see it used in several ways:

**HTML Splicing** At any point where a tag should appear, we can also splice in a Wyvern expression of type `HTML` by enclosing it within curly braces (e.g. on line 13, 15 and 16-19 of Fig. 1). This is implemented on lines 17 and 18 of Fig. 4. The special non-terminal `EXP[T]` signals a switch into parsing a Wyvern expression. The tokenstream will be parsed as a Wyvern expression until a `T` token is encountered *that would otherwise trigger a parse error*. In other words, the Wyvern grammar binds more tightly to itself than to any surrounding TSL. The AST for the parsed Wyvern expression is given an expected type, `HTML`, by simply surrounding it with an ascription (line 18). Because splicing must be structured (a string cannot be concatenated directly), injection and cross-site scripting attacks cannot occur. Safe string splicing (which escapes any inner HTML) could be implemented using another delimiter.

**CSS Splicing** After the `:style` tag appears (e.g. on line 9 of Fig. 1), instead of hard-coding CSS syntax into the HTML DSL, we instead wish to use the TSL associated with a type representing a CSS stylesheet: `CSS`. We do this by again splicing in a Wyvern expression (lines 12-15 of Fig. 4), making sure that it appears in a position where the expected type is `CSS` (the second piece of data associated with the `StyleElement` constructor, in this case). Wyvern is given control until a full expression has been read and an unexpected newline appears (that is, a newline that does not introduce a layout-delimited block).

**Splicing within the CSS TSL** The TSL for `CSS` itself has support for splicing in a similar manner, choosing `%` as the delimiter. It chooses the type based on the semantics of the surrounding CSS form. For example, when a Wyvern expression appears inside

`url`, as on line 10 of Fig. 1, it must be of type `URL`. When a Wyvern expression appears where a color is needed, the `Color` type is used. This type itself has a TSL associated with it that interprets CSS color strings, showing that TSLs can be used within TSLs by simply escaping out to Wyvern, the host language, and then back in. In this case, we emphasize that TSLs produced structured values by calling the `darken` method on it to produce a new color. This method itself takes a `Percentage` as an argument. The TSL for this type accepts literal bodies containing numbers followed by `pct`, or simply a real number without a suffix. These literal bodies, because they begin with a number (and no other form in Wyvern can), does not require delimiters (Fig. 2).

**Splicing within the SQLQuery TSL** The TSL used for SQL queries on line 18 of Fig. 1 follows an identical pattern, allowing strings to be spliced into portions of a query in a safe manner. This prevents SQL injection attacks.

# 3 Syntax

## 3.1 Concrete Syntax

We will now describe the concrete syntax of Wyvern declaratively, using the same layout-sensitive formalism that we have introduced for TSL grammars, developed recently by Adams [1]. Such a formalism is useful because it allows us to implement layout-sensitive syntax, like that we've been describing, without relying on context-sensitive lexers or parsers. Most existing layout-sensitive languages (e.g. Python and Haskell) use hand-rolled context-sensitive lexers or parsers (keeping track of, for example, the indentation level using special `INDENT` and `DEDENT` tokens), but these are more problematic because they cannot be used to generate editor modes, syntax highlighters and other tools automatically. In particular, we will show how the forward references we have described can be correctly encoded without requiring a context-sensitive parser or lexer using this formalism. It is also useful that the TSL for `Parser`, above, uses the same parser technology as the host language, so that it can be used to generate quasiquotes.

Wyvern's concrete syntax, with a few minor omissions for concision, is shown in Figure 6. We first review Adams' formalism in some additional detail, then describe some key features of this syntax.

## 3.2 Background: Adams' Formalism

For each terminal and non-terminal in a rule, Adams proposed associating with them a relational operator, such as =, > and $\geq$ to specify the indentation at which those terms need to be with respect to the non-terminal on the left-hand side of the rule. The indentation level of a term can be identified as the column at which the left-most character of that term appears (not simply the first character, in the case of terms that span multiple lines). The meaning of the comparison operators is akin to their mathematical meaning: = means that the term on the right-hand side has to be at exactly the same indentation as the term on the left-hand side; > means that the term on the right-hand side has to be indented strictly further to the right than the term on the left-hand side; $\geq$ is like >, except the term on the right could also be at the same indentation level as the term on the left-hand side. For example, the production rule of the form $\texttt{A} \rightarrow \texttt{B}^{=}\ \texttt{C}^{\geq}\ \texttt{D}^{>}$ approximately reads as: "Term `B` must be at the same indentation level as term `A`, term `C` may be at the same or a greater indentation level as term `A`, and term `D` must be at an indentation level

```
(* programs *)
p → 'objtype'= ID> NEWLINE> objdecls> metadatadecl> NEWLINE> p=
p → 'casetype'= ID> NEWLINE> casedecls> metadatadecl> NEWLINE> p=
p → e=
metadatadecl → ε | 'metadata'= '='> e>
objdecls → ϵ
objdecls → 'val'= ID> ':'> type NEWLINE> objdecls>
objdecls → 'def'= ID> '('> typelist> ')'> ':'> type> NEWLINE> objdecls>
casedecls → ϵ
casedecls → ID= (ϵ | 'of'> type>) NEWLINE> casedecls>

type → ID= | type= '->'> type> | type= '*'> type>

e → ē=
e → ẽ['~']= NEWLINE> chars>
e → ẽ['new']= NEWLINE> m>
e → ẽ['case(' ē ')']= NEWLINE> r>

(* object definitions *)
m → ε
m → 'val'= ID> '='> e> NEWLINE> m=
m → 'def'= ID> '('> idlist> ')'> '='> e> NEWLINE> d=

(* rules for case analysis (case types and products) *)
r → rc | rp
rc → ID= '('> ID> ')'> '=>'> e>
rc → ID= '('> ID> ')'> '=>'> e> NEWLINE> rc=
rp → '('= idlist> ')'> '=>'> e>

(* expressions containing zero forward references *)
ē → ID=
ē → ē= ':'> type>
ē → 'let'= ID> (ϵ | ':'> type>) '='> e> NEWLINE> ē=
ē → 'fn'= '('> idlist> ')'> (ϵ | ':'> type>) '=>'> ē>
ē → ē= '('> ā̄l> ')'>
ē → '('> ā̄l> ')'>
ē → ē= '.'> ID>
ē → 'toast'= '('> ē> ')'>
ē → 'metadata'= '['> ID> ']'>
ē → inlinelit=
ā̄l → ε | ā̄l_nonempty=
ā̄l_nonempty → ē= | ē= ','> ā̄l_nonempty>
inlinelit → samedelims= | matcheddelims= | numlit=

(* expressions containing exactly one forward reference *)
ẽ[fwd] → fwd=
ẽ[fwd] → ẽ[fwd]= ':'> type>
ẽ[fwd] → 'let'= ID> (ϵ | ':'> type>) '='> e> NEWLINE> ẽ[fwd]=
ẽ[fwd] → 'let'= ID> (ϵ | ':'> type>) '='> ẽ[fwd]> NEWLINE> ē=
ẽ[fwd] → 'fn'= idlist> (ϵ | ':'> type>) '=>'> ẽ[fwd]>
ẽ[fwd] → ẽ[fwd]= '('> ā̄l> ')'>
ẽ[fwd] → ē= '('> ā̃l[fwd]> ')'>
ẽ[fwd] → '('> ā̃l[fwd]> ')'>
ẽ[fwd] → ẽ[fwd]= '.'> ID>
ẽ[fwd] → 'toast'= '('> ẽ[fwd]> ')'>
ā̃l[fwd] → ẽ[fwd]= | ẽ[fwd]= ','> ā̄l_nonempty> | ē= ','> ā̃l[fwd]>
```

Fig. 6: Concrete syntax of TSL Wyvern specified as an Adams grammar. Some standard productions and precedence handling rules have been omitted for concision.

```
1  objtype T
2    val y : HTML
3  let page : HTML->HTML = fn x:HTML => ~
4    :html
5      :body
6        {x}
7  page(case(5 : Nat))
8    Z(_) => (new : T).y
9      val y : HTML = ~
10       :h1 Zero!
11   S(x) => ~
12     :h1 Successor!
```

$$
\begin{array}{l}
\textbf{objtype}\ T\ \{ \\
\quad \textbf{val}\ y : HTML, \\
\quad \textbf{metadata} = (\textbf{new}\ \{\}) : Unit\ \}; \\
(\boldsymbol{\lambda} page : HTML \to HTML. \\
\ page(\textbf{case}(\lfloor 5 \rfloor : Nat)\ \{ \\
\quad Z(\_) \Rightarrow ((\textbf{new}\ \{ \\
\quad\quad \textbf{val}\ y : HTML = \lfloor : h1\ Z! \rfloor\}) : T).y \\
\quad |S(x) \Rightarrow \lfloor : h1\ S! \rfloor\})) \\
\quad (\boldsymbol{\lambda} x : HTML.\ \lfloor : html \\
\quad : body \\
\quad \{x\} \rfloor)
\end{array}
$$

Fig. 7: An example Wyvern program demonstrating forward references. The corresponding abstract syntax, where forward references are inlined, is on the right.

greater than term `A`'s." In particular, if `D` contains a `NEWLINE` character, the next line must be indented past the position of the left-most character of `A` (typically constructed so that it must appear at the beginning of a line). There are no constraints relating `D` to `B` or `C` other than the standard sequencing constraint: the first character of `D` must be further along in the file than the others. Using Adam's formalism, the grammars of real-world languages like Python and Haskell can be written declaratively. This formalism can be integrated into LR and LALR parser generators.

### 3.3 Programs

An example Wyvern program showing several unique syntactic features of TSL Wyvern is shown in Fig. 5. The top level of a program (the `p` non-terminal) consists of a series of type declarations – object types using **`objtype`** or case types using **`casetype`** – followed by an expression, `e`. Each type declaration contains associated declarations – signatures for fields and methods in `objdecls` and case declarations in `casedecls`. Each also can also include a metadata declaration. Metadata is simply an expression associated with the type, used to store TSL logic (and in future work, other logic). Sequences of top-level declarations use the form $\texttt{p}^{=}$ to signify that all the succeeding `p` terms must begin at the same indentation.

### 3.4 Forward Referenced Blocks

Wyvern makes extensive use of forward referenced blocks to make its syntax clean. In particular, layout-delimited TSLs, **new** expressions for creating objects, and the **case** statement for eliminating case types all use forward referenced blocks. Fig. 7 shows all of these in use (assuming suitable definitions of casetypes `Nat` and `HTML`, not included). In the grammar, note particularly the rules for **`let`** and that inline literals, even those containing nested expressions with forward references, can be treated as expressions not containing forward references – *in the initial phase of parsing, before typechecking commences, all literal forms are left unparsed*.

### 3.5 Abstract Syntax

The concrete syntax of a Wyvern program, `p`, is parsed to produce a program in the abstract syntax, $\rho$, shown on the left side of Fig. 8. Forward references are internalized.

$$
\begin{array}{lll}
\rho ::= \theta; e & \tau ::= \mathbf{named}[T] \mid \mathbf{arrow}[\tau, \tau] & \\
\theta ::= \emptyset & & \\
\quad \mid \mathbf{objtype}[T, \omega, e]; \theta & \omega ::= \emptyset \mid \ell[\tau]; \omega & \\
\quad \mid \mathbf{casetype}[T, \chi, e]; \theta & \chi ::= \emptyset \mid C[\tau]; \chi &
\end{array}
$$

$$
\begin{array}{lll}
e ::= x & \hat{e} ::= x & i ::= x \\
\quad \mid \mathbf{easc}[\tau](e) & \quad \mid \mathbf{hasc}[\tau](\hat{e}) & \quad \mid \mathbf{iasc}[\tau](i) \\
\quad \mid \mathbf{elet}(e; x.e) & \quad \mid \mathbf{hlet}(\hat{e}; x.\hat{e}) & \quad \mid \mathbf{ilet}(i; x.i) \\
\quad \mid \mathbf{elam}(x.e) & \quad \mid \mathbf{hlam}(x.\hat{e}) & \quad \mid \mathbf{ilam}(x.i) \\
\quad \mid \mathbf{eap}(e; e) & \quad \mid \mathbf{hap}(\hat{e}; \hat{e}) & \quad \mid \mathbf{iap}(i; i) \\
\quad \mid \mathbf{enew}\,\{m\} & \quad \mid \mathbf{hnew}\,\{\hat{m}\} & \quad \mid \mathbf{inew}\,\{\dot{m}\} \\
\quad \mid \mathbf{eprj}[\ell](e) & \quad \mid \mathbf{hprj}[\ell](\hat{e}) & \quad \mid \mathbf{iprj}[\ell](i) \\
\quad \mid \mathbf{einj}[C](e) & \quad \mid \mathbf{hinj}[C](\hat{e}) & \quad \mid \mathbf{iinj}[C](i) \\
\quad \mid \mathbf{ecase}(e)\,\{r\} & \quad \mid \mathbf{hcase}(\hat{e})\,\{\hat{r}\} & \quad \mid \mathbf{icase}(i)\,\{\dot{r}\} \\
\quad \mid \mathbf{etoast}(e) & \quad \mid \mathbf{htoast}(\hat{e}) & \quad \mid \mathbf{itoast}(i) \\
\quad \mid \mathbf{emetadata}[T] & \quad \mid \mathbf{hmetadata}[T] & \\
\quad \mid \mathbf{lit}[body] & \quad \mid \mathbf{spliced}[e] &
\end{array}
$$

$$
\begin{array}{lll}
m ::= \emptyset & \hat{m} ::= \emptyset & \dot{m} ::= \emptyset \\
\quad \mid \mathbf{eval}[\ell](e); m & \quad \mid \mathbf{hval}[\ell](\hat{e}); \hat{m} & \quad \mid \mathbf{ival}[\ell](i); \dot{m} \\
\quad \mid \mathbf{edef}[\ell](x.e); m & \quad \mid \mathbf{hdef}[\ell](x.\hat{e}); \hat{m} & \quad \mid \mathbf{idef}[\ell](x.i); \dot{m}
\end{array}
$$

$$
\begin{array}{lll}
r ::= \emptyset & \hat{r} ::= \emptyset & \dot{r} ::= \emptyset \\
\quad \mid \mathbf{erule}[C](x.e); r & \quad \mid \mathbf{hrule}[C](x.\hat{e}); \hat{r} & \quad \mid \mathbf{irule}[C](x.i); \dot{r}
\end{array}
$$

Fig. 8: Abstract Syntax of TSL Wyvern programs ($\rho$), type declarations ($\theta$), types ($\tau$), external terms ($e$), translational terms ($\hat{e}$) and internal terms ($i$) and auxiliary forms. Metavariable $T$ ranges over type names, $\ell$ over object member (field and method) labels, $C$ over case labels, $x$ over variables and $body$ over literal bodies. Tuple types are a mode of use of object types, so they are not included in the abstract syntax. For concision, we continue to write pairs as $(i_1, i_2)$ in the rules below.

In particular, note that all literal forms are unified into the abstract literal form $\lfloor body \rfloor$, including the layout-delimited form and number literals. The abstract syntax contains a form, $\mathbf{fromTS}(e)$, that has no analog in the concrete syntax. This will be used internally to ensure hygiene, as we will discuss in the next section.

## 4 Bidirectional Typechecking and Elaboration

We will now specify a type system for the abstract syntax in Fig. 8. Conventional type systems are specified using a typing judgement written like $\Gamma \vdash_\Theta e : \tau$, where the typing context, $\Gamma$, maps bound variables to types, and the named type context, $\Theta$, maps type names to their declarations. Typing judgements do not consider how, when writing a typechecker, it should be considered algorithmically: will a type be provided from the surrounding syntactic context (e.g. when the term appears as a function argument, or an explicit ascription has been provided), so that we simply need to *analyze* $e$ against it, or do we need to *synthesize* a type for $e$ (e.g. when the term appears at the top-level)? Here, this distinction is crucial: a literal can only appear in an analytic context.

*Bidirectional type systems* [27] make this distinction explicit by specifying the type system instead using two simultaneously defined typechecking judgements correspond-

$\boxed{\rho \sim \Theta \rightsquigarrow i : \tau}$ $\quad \Theta ::= \emptyset \mid \Theta, T[\delta, \mu] \quad \delta ::= \ ? \mid \mathbf{ot}[\omega] \mid \mathbf{ct}[\chi] \quad \mu ::= \ ? \mid i : \tau$

$$\frac{\vdash_{\Theta_0} \theta \sim \Theta \quad \emptyset \vdash_{\Theta_0 \Theta} e \rightsquigarrow i \Rightarrow \tau}{\theta; e \sim \Theta \rightsquigarrow i : \tau}\ \textit{Compile}$$

$\boxed{\vdash_\Theta \theta \sim \Theta}$

$$\frac{T \notin \mathrm{dom}(\Theta) \quad \vdash_{\Theta, T[?,?]} \omega \quad \emptyset \vdash_{\Theta, T[\mathbf{ot}[\omega],?]} e_m \rightsquigarrow i_m \Rightarrow \tau_m \quad \vdash_{\Theta, T[\mathbf{ot}[\omega], i_m : \tau_m]} \theta \rightsquigarrow \Theta'}{\vdash_\Theta \mathbf{objtype}[T, \omega, e_m]; \theta \sim T[\mathbf{ot}[\omega], i_m : \tau_m]; \Theta'}\ \textit{OT}$$

$$\frac{T \notin \mathrm{dom}(\Theta) \quad \vdash_{\Theta, T[?,?]} \chi \quad \emptyset \vdash_{\Theta, T[\mathbf{ct}[\chi],?]} e_m \rightsquigarrow i_m \Rightarrow \tau_m \quad \vdash_{\Theta, T[\mathbf{ct}[\chi], i_m : \tau_m]} \theta \rightsquigarrow \Theta'}{\vdash_\Theta \mathbf{casetype}[T, \chi, e_m]; \theta \sim T[\mathbf{ct}[\chi], i_m : \tau_m]; \Theta'}\ \textit{CT}$$

$\boxed{\vdash_\Theta \omega}$ $\dfrac{\ell \notin \mathrm{dom}(\omega) \quad \vdash_\Theta \tau \quad \vdash_\Theta \omega}{\vdash_\Theta \ell[\tau]; \omega}$ *M-decl* $\quad \boxed{\vdash_\Theta \chi}$ $\dfrac{C \notin \mathrm{dom}(\chi) \quad \vdash_\Theta \tau \quad \vdash_\Theta \chi}{\vdash_\Theta C[\tau]; \chi}$ *C-decl*

$\boxed{\vdash_\Theta \tau}$ $\dfrac{T[\delta, \mu] \in \Theta}{\vdash_\Theta \mathbf{named}[T]}$ *Ty-named* $\quad \dfrac{\vdash_\Theta \tau_1 \quad \vdash_\Theta \tau_2}{\vdash_\Theta \mathbf{arrow}[\tau_1, \tau_2]}$ *Ty-arrow*

Fig. 9: Typechecking and elaboration of programs, $\rho$. Note that type declarations can only be recursive, not mutually recursive, with these rules. The prelude $\Theta_0$ (see Fig. 5) defines mutually recursive types, so we cannot write a $\theta_0$ corresponding to $\Theta_0$ given the rules above. For concision, the rules to support mutual recursion as well as omitted rules for empty declarations are available in a technical report [**?**].

ing to these two situations. For TSL Wyvern, we need to also simultaneously perform an elaboration of the external language, which contains literals, to an "internal language", $i$, the syntax for which is shown on the right side of Fig. 8. The internal language does not have literals, nor a form for accessing the metadata of a named type explicitly (the elaboration process inserts the statically known metadata value, tracked by the named type context, directly). The judgement $\Gamma \vdash_\Theta e \rightsquigarrow i \Rightarrow \tau$ means that under typing context $\Gamma$ and named type context $\Theta$, external term $e$ elaborates to internal term $i$ and synthesizes type $\tau$. The judgement $\Gamma \vdash_\Theta e \rightsquigarrow i \Leftarrow \tau$ is analagous but for situations where we are analyzing $e$ against type $\tau$. This manner of specifying a type-directed mapping from external terms to a smaller collection internal terms, which are the only terms that are given a dynamic semantics, is stylistically related to the Harper-Stone elaboration semantics for Standard ML [10] so our semantics for TSL Wyvern is a form of *bidirectionally typed elaboration semantics*.

### 4.1 Programs and Type Declarations

Before considering these judgements in detail, let us briefly discuss the steps leading up to typechecking and elaboration of the top-level term, specified by the compilation judgement, $\rho \sim \Theta \rightsquigarrow i : \tau$, defined in Fig. 9. We first load the prelude, $\Theta_0$ (see Fig. 5), then validate the provided user-defined type declarations, $\theta$, to produce a corresponding named typed context, $\Theta$. During this process, we synthesize a type for the associated metadata terms (under the empty typing context) and store their elaborations in the type context $\Theta$ (we do not evaluate the elaboration to a value immediately here, though in a language with effects, the choice of when to evaluate the term is important). Note

that type names must be unique (we plan to use a URI-based mechanism in practice). Finally, the top-level external term must synthesize a type $\tau$ and produces an elaboration $i$ under an empty typing context and a named type context combining the prelude with the named type context induced by the user-defined types, written $\Theta_0\Theta$.

### 4.2 External Terms

The bidirectional typechecking and elaboration rules for external terms are shown beginning in Fig. 10. Nearly all the rules are standard for simply typed lambda calculus with labeled sums and labeled products, and the elaborations are direct. We refer the reader to standard texts on type systems (e.g. [9]) to understand the basic constructs, and to course material[1] on bidirectional typechecking for background. In our presentation, all introductory forms are analytic and all elimination forms are synthetic.

The introductory form for object types, **enew** $\{m\}$, prevents the manual introduction of parse streams (only the semantics can introduce parse streams, to permit us to enforce hygiene, as we will discuss below). The auxiliary judgement $\Gamma \vdash^{T}_{\Theta} m \rightsquigarrow \dot{m} \Leftarrow \omega$ analyzes the member definitions $m$ against the member declarations $\omega$ while rewriting them to the internal member definitions, $\dot{m}$. Method definitions involve a self-reference, so the judgement keeps track of the type name, $T$. We implicitly assume that member definitions and declarations are congruent up to reordering.

The introduction form for case types is written **einj**$[C](e)$, where $C$ is the case name and $e$ is the associated data. The type of the data associated with each case is stored in the case type's declaration, $\chi$. Because the introductory form is analytic, multiple case types can use the same case names (unlike in, for example, ML). The elimination form, **ecase**$(e)\,\{r\}$, performs simple exhaustive case analysis (we leave support for nested pattern matching as future work) using the auxiliary judgement $\Gamma \vdash_{\Theta} r \rightsquigarrow \dot{r} \Leftarrow \chi \Rightarrow \tau$, which checks that each case in $\chi$ appears in a rules in the rule sequence $r$, rewriting it to the internal rule sequence $\dot{r}$. Every rule must synthesize the same type, $\tau$.

The rule *T-metadata* shows how the appropriate metadata is extracted from the named type context and inserted directly in the elaboration. We will return to the rule *T-toast* when discussing hygiene.

### 4.3 Literals

In the example in Fig. 4, we showed a TSL being defined using a parser generator based an Adams grammars. As we noted, a parser generator can itself be seen as a TSL for a parser, and a parser is the fundamental construct that becomes associated with a type to form a TSL. The declaration for the prelude type `Parser`, shown in Fig. 5, shows that it is an object type with a `parse` function taking in a `ParseStream` and producing a `Result`, which is a case type that indicates either that parsing succeeded, in which case an elaboration of type `Exp` is paired with the remaining parse stream (to allow one parser to call another), or that parsing failed, in which case an error message and location is provided. This function is called by the typechecker when analyzing the literal form, as shown in the key rule of our system, *T-lit*, shown in Fig. 11. Note that we do not explicitly handle failure in the specification, but in practice we would use the data provided for the failure case to report to the user.

[1] `http://www.cs.cmu.edu/~fp/courses/15312-f04/handouts/15-bidirectional.pdf`

$\boxed{\Gamma \vdash_\Theta e \rightsquigarrow i \Rightarrow \tau}$ $\boxed{\Gamma \vdash_\Theta e \rightsquigarrow i \Leftarrow \tau}$ $\quad \Gamma ::= \emptyset \mid \Gamma, x : \tau$

$$\frac{\Gamma \vdash_\Theta e \rightsquigarrow i \Rightarrow \tau}{\Gamma \vdash_\Theta e \rightsquigarrow i \Leftarrow \tau}\ \textit{T-syn-to-ana} \qquad \frac{\vdash_\Theta \tau \quad \Gamma \vdash_\Theta e \rightsquigarrow i \Leftarrow \tau}{\Gamma \vdash_\Theta \textbf{easc}[\tau](e) \rightsquigarrow \textbf{iasc}[\tau](i) \Rightarrow \tau}\ \textit{T-asc}$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash_\Theta x \rightsquigarrow x \Rightarrow \tau}\ \textit{T-var} \qquad \frac{\Gamma \vdash_\Theta e_1 \rightsquigarrow i_1 \Rightarrow \tau_1 \quad \Gamma, x : \tau_1 \vdash_\Theta e_2 \rightsquigarrow i_2 \Rightarrow \tau}{\Gamma \vdash_\Theta \textbf{elet}(e_1; x.e_2) \rightsquigarrow \textbf{ilet}(i_1; x.i_2) \Rightarrow \tau}\ \textit{T-let}$$

$$\frac{\Gamma, x : \tau_1 \vdash_\Theta e \rightsquigarrow i \Leftarrow \tau_2}{\Gamma \vdash_\Theta \textbf{elam}(x.e) \rightsquigarrow \textbf{ilam}(x.i) \Leftarrow \textbf{arrow}[\tau_1, \tau_2]}\ \textit{T-abs}$$

$$\frac{\Gamma \vdash_\Theta e_1 \rightsquigarrow i_1 \Rightarrow \tau_1 \to \tau_2 \quad \Gamma \vdash_\Theta e_2 \rightsquigarrow i_2 \Leftarrow \tau_1}{\Gamma \vdash_\Theta \textbf{eap}(e_1; e_2) \rightsquigarrow \textbf{iap}(i_1; i_2) \Rightarrow \tau_2}\ \textit{T-ap}$$

$$\frac{T \neq \mathit{ParseStream} \quad T[\textbf{ot}[\omega], \mu] \in \Theta \quad \Gamma \vdash^T_\Theta m \rightsquigarrow \dot{m} \Leftarrow \omega}{\Gamma \vdash_\Theta \textbf{enew}\,\{m\} \rightsquigarrow \textbf{inew}\,\{\dot{m}\} \Leftarrow \textbf{named}[T]}\ \textit{T-new}$$

$$\frac{\Gamma \vdash_\Theta e \rightsquigarrow i \Rightarrow \textbf{named}[T] \quad T[\textbf{ot}[\omega], \mu] \in \Theta \quad \ell[\tau] \in \omega}{\Gamma \vdash_\Theta \textbf{eprj}[\ell](e) \rightsquigarrow \textbf{iprj}[\ell](i) \Rightarrow \tau}\ \textit{T-prj}$$

$$\frac{T[\textbf{ct}[\chi], \mu] \in \Theta \quad C[\tau] \in \chi \quad \Gamma \vdash_\Theta e \rightsquigarrow i \Leftarrow \tau}{\Gamma \vdash_\Theta \textbf{einj}[C](e) \rightsquigarrow \textbf{iinj}[C](i) \Leftarrow \textbf{named}[T]}\ \textit{T-inj}$$

$$\frac{\Gamma \vdash_\Theta e \rightsquigarrow i \Rightarrow \textbf{named}[T] \quad T[\textbf{ct}[\chi], \mu] \in \Theta \quad \Gamma \vdash_\Theta r \rightsquigarrow \dot{r} \Leftarrow \chi \Rightarrow \tau}{\Gamma \vdash_\Theta \textbf{ecase}(e)\,\{r\} \rightsquigarrow \textbf{icase}(i)\,\{\dot{r}\} \Rightarrow \tau}\ \textit{T-case}$$

$$\frac{\Theta_0 \subset \Theta \quad \Gamma \vdash_\Theta e \rightsquigarrow i \Rightarrow \tau}{\Gamma \vdash_\Theta \textbf{etoast}(e) \rightsquigarrow \textbf{itoast}(i) \Rightarrow \textbf{named}[\mathit{Exp}]}\ \textit{T-toast}$$

$$\frac{T[\delta, i : \tau] \in \Theta}{\Gamma \vdash_\Theta \textbf{emetadata}[T] \rightsquigarrow i \Rightarrow \tau}\ \textit{T-metadata}$$

$\boxed{\Gamma \vdash^T_\Theta m \rightsquigarrow \dot{m} \Leftarrow \omega}$

$$\frac{}{\Gamma \vdash^T_\Theta \emptyset \rightsquigarrow \emptyset \Leftarrow \emptyset}\ \textit{T-unit}$$

$$\frac{\Gamma \vdash_\Theta e \rightsquigarrow i \Leftarrow \tau \quad \Gamma \vdash^T_\Theta m \rightsquigarrow \dot{m} \Leftarrow \omega}{\Gamma \vdash^T_\Theta \textbf{eval}[\ell](e); m \rightsquigarrow \textbf{ival}[\ell](i); \dot{m} \Leftarrow \ell[\tau]; \omega}\ \textit{T-val}$$

$$\frac{\Gamma, x : \textbf{named}[T] \vdash_\Theta e \rightsquigarrow i \Leftarrow \tau \quad \Gamma \vdash^T_\Theta m \rightsquigarrow \dot{m} \Leftarrow \omega}{\Gamma \vdash^T_\Theta \textbf{edef}[\ell](x.e); m \rightsquigarrow \textbf{idef}[\ell](x.i); \dot{m} \Leftarrow \ell[\tau]; \omega}\ \textit{T-def}$$

$\boxed{\Gamma \vdash_\Theta r \rightsquigarrow \dot{r} \Leftarrow \chi \Rightarrow \tau}$

$$\frac{}{\Gamma \vdash_\Theta \emptyset \rightsquigarrow \emptyset \Leftarrow \emptyset \Rightarrow \tau}\ \textit{T-void}$$

$$\frac{\Gamma, x : \tau_1 \vdash_\Theta e \rightsquigarrow i \Rightarrow \tau_2 \quad \Gamma \vdash_\Theta r \rightsquigarrow \dot{r} \Leftarrow \chi \Rightarrow \tau_2}{\Gamma \vdash_\Theta \textbf{erule}[C](x.e); r \rightsquigarrow \textbf{irule}[C](x.i); \dot{r} \Leftarrow C[\tau_1]; \chi \Rightarrow \tau_2}\ \textit{T-rule}$$

Fig. 10: Statics for external terms, $e$. The rule for literals is shown in Fig. 11.

$$\frac{\begin{array}{c}\Theta_0 \subset \Theta \quad T[\delta, i_m : HasTSL] \in \Theta \quad \texttt{parsestream}(body) = i_{ps} \\ \textbf{iap}(\textbf{iprj}[parse](\textbf{iprj}[parser](i_m)); i_{ps}) \Downarrow \textbf{iinj}[OK]((i_{ast}, i'_{ps})) \\ i_{ast} \uparrow \hat{e} \quad \Gamma; \emptyset \vdash_\Theta \hat{e} \rightsquigarrow i \Leftarrow \textbf{named}[T]\end{array}}{\Gamma \vdash_\Theta \textbf{lit}[body] \rightsquigarrow i \Leftarrow \textbf{named}[T]} \textit{ T-lit}$$

Fig. 11: Statics for external terms, $e$, continued. This is the key rule (see text).

The rule *T-lit* operates as follows:

1. This rule requires that the prelude is available. For technical reasons, we include a check that the prelude was actually included in the named type context.
2. The metadata of the type the literal is being checked against, which must be of type $HasTSL$, is extracted from the named type context. Note that in a language with subtyping or richer forms of type equality, which would be necessary for situations where the metadata might serve other roles, the check that $i_m$ defines a TSL would require an additional premise.
3. A parse stream, an internal term of type `ParseStream`, $i_{ps}$, is generated from the body of the literal. This type is an object that that allows the reading of tokens, as well as additional methods, discussed further below.
4. The `parse` method is called with this parse stream. If it evaluates to a reified elaboration, $i_{ast}$ (of type `Exp`) and a remaining parse stream, $i_{ps}$, then parsing was successful. Note that we use shorthand for pairs in the rule for concision, and the relation $i \Downarrow i'$ defines evaluation to a value (see the caption of Fig. 15).
5. The reified elaboration is *dereified* into a corresponding *translational term*, $\hat{e}$, as specified in Fig. 12. The syntax for translational terms mirrors that of external terms, but does not include literal forms. It adds the form **spliced**$[e]$, representing an external term spliced into a literal body.
   The key rule is *U-Spl* – the only way to generate a translational term of this form is by asking for (a portion of) a parse stream to be parsed as a Wyvern expression or identifier. The reified form, unlike the translational form it corresponds to, does not contain the expression itself, but rather just a portion of the parse stream that should be recognized. Because parse streams (and thus portions thereof) can originate only metatheoretically, we know that $e$ must be an external term written concretely by the TSL client in the body of the literal being analyzed. This is key to guaranteeing hygiene in the final step.
   The prelude methods `parse_exp` and `parse_id` return a value having this reified form corresponding to the first external term found in the parse stream (but, as just described, not necessarily the term itself) paired with the remainder of the parse stream. These methods themselves are not treated specially by the compiler but, for convenience, are associated with `ParseStream`.
6. The final step is to typecheck and elaborate this translational term. This involves the bidirectional typing judgements shown in Fig. 14. This judgement has a form similar to that for external terms, but with the addition of an "outer typing context", written $\Gamma_{\text{out}}$ in the rules. This holds the context that the literal appeared in, so that the "main" typing context can be emptied to ensure that elaborations are closed

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

$$\frac{i_{id} \uparrow x}{\textbf{iinj}[Var](i_{id}) \uparrow x}\ \textit{U-Var}$$

$$\frac{i_1 \uparrow \tau \quad i_2 \uparrow \hat{e}}{\textbf{iinj}[Asc]((i_1, i_2)) \uparrow \textbf{hasc}[\tau](\hat{e})}\ \textit{U-Asc}$$

$$\frac{i_{id} \uparrow x \quad i \uparrow \hat{e}}{\textbf{iinj}[Lam]((i_{id}, i)) \uparrow \textbf{hlam}(x.\hat{e})}\ \textit{U-Lam}$$

$$\frac{i_1 \uparrow \hat{e}_1 \quad i_2 \uparrow \hat{e}_2}{\textbf{iinj}[Ap]((i_1, i_2)) \uparrow \textbf{hap}(\hat{e}_1, \hat{e}_2)}\ \textit{U-Ap}$$

$$\cdots$$

$$\frac{\texttt{body}(i_{ps})\texttt{=}body \quad \texttt{eparse}(body)\texttt{=}e}{\textbf{iinj}[Spliced](i_{ps}) \uparrow \textbf{spliced}[e]}\ \textit{U-Spl}$$

$\boxed{i \uparrow \tau}$

$$\frac{i_{name} \uparrow T}{\textbf{iinj}[Named](i_{name}) \uparrow \textbf{named}[T]}\ \textit{U-N}$$

$$\frac{i_1 \uparrow \tau_1 \quad i_2 \uparrow \tau_2}{\textbf{iinj}[Arrow]((i_1, i_2)) \uparrow \textbf{arrow}[\tau_1, \tau_2]}\ \textit{U-A}$$

Fig. 12: Dereification rules, used by rule *T-lit* (above) to determine the translational term encoded by the internal term of type **named**$[Exp]$.

$\boxed{i \downarrow i}$

$$\frac{x \downarrow i_{id}}{x \downarrow \textbf{iinj}[Var](i_{id})}\ \textit{R-Var}$$

$$\frac{\tau \downarrow i_1 \quad i \downarrow i_2}{\textbf{iasc}[\tau](i) \downarrow \textbf{iinj}[Asc]((i_1, i_2))}\ \textit{R-Asc}$$

$$\frac{x \downarrow i_{id} \quad i \downarrow i'}{\textbf{ilam}(x.i) \downarrow \textbf{iinj}[Lam]((i_{id}, i'))}\ \textit{R-Lam}$$

$$\frac{i_1 \downarrow i'_1 \quad i_2 \downarrow i'_2}{\textbf{iap}(i_1; i_2) \downarrow \textbf{iinj}[Ap]((i'_1, i_2))}\ \textit{R-Ap}$$

$$\cdots$$

$\boxed{\tau \downarrow i}$

$$\frac{T \downarrow i_{name}}{\textbf{named}[T] \downarrow \textbf{iinj}[Named](i_{name})}\ \textit{R-N}$$

$$\frac{\tau_1 \downarrow i_1 \quad \tau_2 \downarrow i_2}{\textbf{arrow}[\tau_1, \tau_2] \downarrow \textbf{iinj}[Arrow]((i_1, i_2))}\ \textit{R-A}$$

Fig. 13: Reification rules, used by the **itoast** ("to AST") operator (Fig. 15) to permit generating an internal term of type **named**$[Exp]$ corresponding to the value of the argument (a form of serialization).

except for portions derived from the parse stream. Each rule in Fig. 10 should be thought of as having a corresponding rule in Fig. 14. Two examples are shown for concision. The outer context is threaded two opaquely in all cases except the rule for spliced external terms. We discuss these rules further below.

### 4.4 Hygiene

A concern with any term rewriting system is *hygiene* – how should variables in the generated AST be bound? In particular, if the rewriting system generates an *open term*, then it is making assumptions about the names of variables in scope at the site where the TSL is being used, which is incorrect. Those variables should only be identifiable up to alpha renaming. Only the *user* of a TSL knows which variables are in scope. The strictest rule would simply reject all open terms, but this would prevent even spliced terms written by the TSL client, who presumably is aware of variable bindings at the use site, from referring to local variables. Moreover, the variables in these terms should be bound to what the client expects. The elaboration should not be able to surreptitiously or accidentally shadow variables in spliced terms that may be otherwise bound at the use site (e.g. variables named `tmp`).

The solution to both of these issues, which we have outlined above, is quite simple: we construct the system so that we know which sub-terms originate from the TSL

$\boxed{\Gamma; \Gamma \vdash_\Theta \hat{e} \rightsquigarrow i \Rightarrow \tau}$ $\boxed{\Gamma; \Gamma \vdash_\Theta \hat{e} \rightsquigarrow i \Leftarrow \tau}$

$$\frac{x : \tau \in \Gamma}{\Gamma_{\text{out}}; \Gamma \vdash_\Theta x \rightsquigarrow x \Rightarrow \tau}\ \textit{H-var} \qquad \frac{\Gamma_{\text{out}}; \Gamma, x : \tau_1 \vdash_\Theta \hat{e} \rightsquigarrow i \Leftarrow \tau_2}{\Gamma_{\text{out}}; \Gamma \vdash_\Theta \textbf{hlam}(x.\hat{e}) \rightsquigarrow \textbf{ilam}(x.i) \Leftarrow \textbf{arrow}[\tau_1, \tau_2]}\ \textit{H-abs}$$

$$\cdots$$

$$\frac{\Gamma_{\text{out}} \vdash_\Theta e \rightsquigarrow i \Leftarrow \tau}{\Gamma_{\text{out}}; \Gamma \vdash_\Theta \textbf{spliced}[e] \rightsquigarrow i \Leftarrow \tau}\ \textit{H-spl-A} \qquad \frac{\Gamma_{\text{out}} \vdash_\Theta e \rightsquigarrow i \Rightarrow \tau}{\Gamma_{\text{out}}; \Gamma \vdash_\Theta \textbf{spliced}[e] \rightsquigarrow i \Rightarrow \tau}\ \textit{H-spl-S}$$

Fig. 14: Statics for translational terms, $\hat{e}$. Each rule in Fig. 10 corresponds to an analagous rule here by threading the outer context through opaquely (e.g. the rules for variables and functions, shown here). The outer context is only used by the rules for **spliced**$[e]$, representing external terms that were spliced into TSL bodies. Only these terms can access outer variables, achieving hygiene (see text). Note that elaboration is implicitly capture-avoiding here (we assume unique names for internal variables can be generated whenever necessary, see Sec. 6).

$\boxed{i \mapsto i}$ $\quad \cdots \quad$ $\dfrac{i \mapsto i'}{\textbf{itoast}(i) \mapsto \textbf{itoast}(i')}\ \textit{D-Toast-1}$ $\qquad \dfrac{i\ \texttt{val} \quad i \downarrow i'}{\textbf{itoast}(i) \mapsto i'}\ \textit{D-Toast-2}$

Fig. 15: Dynamics for internal terms, $i$. Only internal terms have a dynamic semantics. Most constructs in TSL Wyvern are standard and omitted, as our focus in this paper is on the statics. The only novel internal form, **itoast**$(i)$, extracts an AST (of type **named**$[Exp]$) from the value of $i$, shown.

client, marking them as **spliced**$[e]$. These terms can refer only to variables in the client's context, $\Gamma_{\text{out}}$, as seen in the premises of the two rules for this form (one for analysis, one for synthesis). The remainder of the term is generated by the TSL provider, so it can refer only to variables introduced earlier in the elaboration, tracked by the context $\Gamma$. The two are kept separate. If the TSL wishes to introduce variables in spliced terms, it must do so by via a function application (as in the TSL for `Parser` discussed earlier), ensuring that the client has full control over variable binding.

### 4.5 From Values to ASTs

In some rewriting systems, free variables become bound to their values at the generation site, rather than the use site. In the formulation just discussed, this does not directly occur – all free variables lead to errors when returned by a TSL definition. To permit lifting values bound at the generation site to ASTs for use at the use site, we include the primitive operator `toast(e)`. This simply takes the value of `e` and reifies it, producing a term of type `Exp`, as specified in Figs. 15 and Fig. 13. This can be used to "bake in" a value known at compile time into the generated code safely. The rules for reification, used here, and dereification, used in the literal rule described above, are notionally dual.

The TSL associated with `Exp`, implementing quasiquotes, can perform a free variable analysis and insert these terms automatically (by itself treating the free variables as spliced terms), so they are only explicitly needed when generating an AST manually.

$$\boxed{\Gamma \vdash_\Theta i \Rightarrow \tau} \quad \boxed{\Gamma \vdash_\Theta i \Leftarrow \tau} \quad \cdots \quad \frac{T[\mathbf{ot}[\omega], \mu] \in \Theta \quad \Gamma \vdash^T_\Theta \dot{m} \Leftarrow \omega}{\Gamma \vdash_\Theta \mathbf{inew}\,\{\dot{m}\} \Leftarrow \mathbf{named}[T]}\ \textit{IT-new}$$

Fig. 16: Statics for internal terms, $i$. Each rule in Fig. 10 corresponds to an analogous rule here by removing the elaboration portion. Only the rule for object introduction differs, in that we no longer restrict the introduction of parse streams (internal terms are never written directly by users of the language).

### 4.6 Safety

The semantics we have defined constitute a type safe language. There are two key theorems from which type safety follows directly: type safety of the internal language, and type preservation of the elaboration process.

To prove internal type safety, we define a bidirectional typing judgement for the internal language, shown and described in Fig. 16. We also define a well-formedness judgement for named type contexts (shown in the accompanying technical report [**?**]).

**Theorem 1 (Internal Type Safety).** *If* $\vdash \Theta$ *and* $\emptyset \vdash_\Theta i \Leftarrow \tau$ *or* $\emptyset \vdash_\Theta i \Rightarrow \tau$*, then either* $i$ `val` *or* $i \mapsto i'$ *such that* $\emptyset \vdash_\Theta i' \Leftarrow \tau$*.*

*Proof.* The dynamics, which we omit for concision, are standard, so the proof is by a standard preservation and progress argument. The only interesting case of the proof involves $\mathbf{etoast}(e)$, for which we need the following lemma.

**Lemma 1 (Reification).** *If* $\Theta_0 \subset \Theta$ *and* $\emptyset \vdash_\Theta i \Leftarrow \tau$ *then* $i \downarrow i'$ *and* $\emptyset \vdash_\Theta i' \Leftarrow \textbf{\textit{named}}[Exp]$*.*

*Proof.* The proof is by a straightforward induction over the reification rules. Auxiliary lemmas about reification of identifiers and types are similarly straightforward. □

If the elaboration of a closed, well-typed external term generates an internal term of the same type, then internal type safety implies that evaluation will not go wrong, achieving type safety. We generalize this argument about type preservation to arbitrary typing contexts by defining an auxiliary well-formedness judgement for contexts (shown in the technical report). The relevant theorem is below:

**Theorem 2 (External Type Preservation).** *If* $\vdash \Theta$ *and* $\vdash_\Theta \Gamma$ *and* $\Gamma \vdash_\Theta e \rightsquigarrow i \Leftarrow \tau$ *or* $\Gamma \vdash_\Theta e \rightsquigarrow i \Rightarrow \tau$ *then* $\Gamma \vdash_\Theta i \Leftarrow \tau$*.*

*Proof.* We proceed by inducting over the the typing derivations. Nearly all the elaborations are direct, so the proof is by straightforward applications of induction hypotheses. The only cases of note are:

- $e = \mathbf{enew}\,\{m\}$. Here the corresponding rule for the elaboration is identical but more permissive, so the induction hypothesis applies.
- $e = \mathbf{emetadata}[T]$. Here, the elaboration generates the metadata value directly. Well-formedness of $\Theta$ implies that the metadata term is of the type assigned.
- $e = \mathbf{lit}[body]$. Here, we need to apply internal type safety as well as a mutually defined type preservation lemma about translational terms, below.

**Lemma 2 (Translational Type Preservation).** *If* $\vdash \Theta$ *and* $\vdash_\Theta \Gamma_{out}$ *and* $\vdash_\Theta \Gamma$ *and* $dom(\Gamma_{out}) \cap dom(\Gamma) = \emptyset$ *(which we can assume implicitly by alpha renaming at binding sites) and* $\Gamma_{out}; \Gamma \vdash_\Theta \hat{e} \rightsquigarrow i \Leftarrow \tau$ *or* $\Gamma_{out}; \Gamma \vdash_\Theta \hat{e} \rightsquigarrow i \Rightarrow \tau$ *then* $\Gamma_{out}\Gamma \vdash_\Theta i \Leftarrow \tau$.

*Proof.* The proof follows the same argument as above. The outer context is threaded through opaquely by the inductive hypothesis. The only rules of note are 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 external term written in concrete syntax, since we know it was derived from a portion of the literal body. □

Putting these definitions and theorems together, we can prove the correctness of compilation theorem below. This plus internal type safety constitutes type safety for the language as a whole.

**Theorem 3 (Compilation).** *If* $\rho \sim \Theta \rightsquigarrow i : \tau$ *then* $\vdash \Theta$ *and* $\emptyset \vdash_\Theta i \Leftarrow \tau$.

### 4.7 Decidability

Because we are executing user-defined parsers during typechecking, we do not have a straightforward statement of decidability (i.e. termination) of typechecking. The parser might not terminate. Non-decidability is strictly due to user-defined parsing code. Typechecking of programs that do not contain literals is guaranteed to terminate, as is typechecking of $\hat{e}$ and $i$ (which we do not actually need to do in practice by Theorem 1). Termination of parsers and parser generators has previously been studied (e.g. [16]) and the techniques can be applied to user-defined parsing code to increase confidence in termination. Few compilers, even those with high demands for correctness (e.g. CompCert [18]), have made it a priority to fully verify and prove termination of the parser. This is because it is perceived that most bugs in compilers arise due to incorrect optimization passes, not initial parsing and elaboration logic.

## 5 Corpus Analysis

We performed a corpus analysis on existing Java code to assess how frequently there are opportunities to use TSLs. As a proxy for this goal, we examined `String` arguments passed into Java constructors, for two reasons:

1. The `String` type may be used to represent a large variety of notations, many of which may be expressed using TSLs.
2. We hypothesized that opportunities to use TSLs would often come when initializing an object with state described by the TSL.

*Methodology.* We ran our analysis on a recent version (20130901r) of the Qualitas Corpus [32], consisting of 107 Java projects, and searched for constructors that used `String`s that could be substituted with TSLs. To perform the search, we used command line tools, such as `grep` and `sed`, and a text editor features such as search and substitution. After we found the constructors, we chose those that took at least one `String` as an argument. Via a visual scan of the names of the constructors and their `String` arguments, we inferred how the constructors and the arguments were intended to be used.

*Results.* We found 124,873 constructors and that 19,288 (15%) of them could use TSLs. Table 1 gives more details on types of `String` arguments we found that could be substituted with TSLs. The "Identifier" category comprises process IDs, user IDs, column or row IDs, etc. that usually must be unique; the "Pattern" category includes regular expressions, prefixes and suffixes, delimiters, format templates, etc.; the "Other" category contains `String`s used for ZIP codes, passwords, queries, IP addresses, versions, HTML and XML code, etc.; and the "Directory path" and "URL/URI" categories are self-explanatory.

| **Type of String** | **Number** | **Percentage** |
|---|---|---|
| Identifier | 15,642 | 81% |
| Directory path | 823 | 4% |
| Pattern | 495 | 3% |
| URL/URI | 396 | 2% |
| Other (ZIP code, password, query, HTML/XML, IP address, version, etc.) | 1,932 | 10% |
| **Total:** | **19,288** | **100%** |

Table 1: Types of `String` arguments in Java constructors that could use TSLs

*Limitations.* There are three limitations to our corpus analysis. First, the proxy that we chose for finding how often TSLs could be used in existing Java code is imprecise. Our corpus analysis focused exclusively on Java constructors and thus did not consider other programming constructs, such as method calls, variable assignments, etc., that could possibly use TSLs. Also, there are other datatypes, not just `String`, that could be substituted with TSLs, e.g. `Path` and `URL`. Second, our search for constructors with the use of command line tools and text editor features may not have identified all the Java constructors present in the corpus. Finally, the inference of the intended functionality of the constructor and the passed in `String` argument was based on the authors' programming experience and was thus subjective.

Despite the limitations of our corpus analysis, it shows that there is the potential to enhance existing code with type-specific languages since numerous `String`s could be substituted with TSLs and a significant portion of Java constructors could take advantage of this fact.

## 6 Implementation

The implementation of Wyvern is based around a core parsing and typechecking system, with TSL parsers being added as an an intermediate step. The top-level parser for Wyvern is produced by the Copper parser generator [35] and uses stateful LALR parsing to handle whitespace and the core language, as well as for inline TSL invocations. Forward references, such as the TSL tilde, the new keyword, and case statements, are handled using a special "signal" token, where the parser generates the signal if it reaches the end of an expression containing the forward reference. When the parser then

encounters this signal token, it enters an appropriate state depending on the type of forward reference encountered. TSL blocks are handled as strings, preserving whitespace, and new and case statements are parsed using their respective grammars.

Wyvern performs TSL parsing as part of its typechecker, which is otherwise a standard bidirectional typesystem implementation. When the typechecker encounters a TSL block, it retrieves the associated parser and applies it to the string produced by the first stage of parsing. These steps may then be performed recursively, if the TSL parser requires that additional Wyvern code be parsed inside itself.

As of this writing, the implementation of the TSL parser extension mechanism is still incomplete, but does support extending Wyvern with simple TSLs such as a simple calculator language.

## 7 Related Work

Closely related to our approach of type-driven parsing is a concurrent paper by Ichikawa et al. [11] that presents *protean operators*. The paper describes the *ProteaJ* language, based on Java, which allows a programmer to define flexible operators together with named types for any nonterminal in the grammar. These named types are then used to allow safe composition of different languages defined using protean operators where any syntactic conflict is resolved by looking at the expected type. Conflicts may still arise when the expected type matches two protean operators; in this case PoroteaJ allows the programmer to define a precedence between operators. In contrast, by associating parsers with types, our approach avoids all conflicts, achieving a stricter notion of modularity at the cost of some extensibility.

Language macros are the most explored way of extending programming languages, with Scheme and other Lisp-style languages' hygienic macros being the 'gold standard.' In those languages, macros are written in the language itself and benefit from the simple syntax – parentheses universally serve as expression delimiters (although proposals for whitespace as a substitute for parentheses have been made [21]). Our work is inspired by this flexibility, but aims to support richer syntax as well as static types. Wyvern's use of types to trigger parsing avoids the overhead of invoking macros explicitly by name, and makes it easier to compose TSLs declaratively.

Another way to approach language extensibility is to go a level of abstraction above parsing, as is done via metaprogramming facilities. For instance, OJ (previously, OpenJava) [31] provides a macro system based on a meta-object protocol, and Backstage Java [26], Template Haskell [29] and Converge [33] employ compile-time metaprogramming. Each of these systems provide macro-style rewriting of source code, but they provide at most limited extension of language parsing.

Other systems aim at providing forms of syntax extension that change the host language, as opposed to our whitespace-delimited approach. For example, Camlp4 [4] is a preprocessor for OCaml that can be used to extend the concrete syntax of the language with parsers and extensible grammars. SugarJ [6] supports syntactic extension of the Java language by adding libraries. Wyvern differs from these approach in that the core language is not extended directly, so conflicts cannot arise at link-time.

Scoping TSLs to expressions of a single type comes at the expense of some flexibility, but we believe that many uses of domain-specific languages are of this form already. A previous approach has considered type-based disambiguation of parse forests

for supporting quotation and anti-quotation of arbitrary object languages [2]. Our work is similar in spirit, but does not rely on generation of parse forests and associates grammars with types, rather than types with grammar productions. This provides stronger modularity guarantees and is arguably simpler. C# expression trees [20] are similar in that, when the type of a term is `Expression<T->T'>`, it is parsed as a quotation. However, like the work just mentioned, this is *specifically* to support quotations. Our work supports quotations in addition to a variety of other work.

Many approaches to syntax extension, such as XJ [3] are keyword-directed in some form. We believe that a type-directed approach is more seamless and general, sacrificing a small amount of identifiability in some cases.

In terms of work on safe language composition, Schwerdfeger and van Wyk [28] proposed a solution that make strong safety guarantees provided that the languages comply with certain grammar restrictions, concerning first and follow sets of the host language and the added new languages. It also relied on strongly named entry tokens, like keyword delimited approaches. Our approach does not impose any such restrictions while still making safety guarantees.

Domain-specific language frameworks and language workbenches, such as Spoofax [14], Ensō [19] and others [15, 34], also provide a possible solution for the language extension task. They provide support for generating new programming languages and tooling in a modular manner. The Marco language [17] similarly provides macro definition at a level of abstraction that is largely independent of the target language. In these approaches, each TSL is *external* relative to the host language; in contrast, Wyvern focuses on extensibility *internal* to the language, improving interoperability and composability.

Ongoing work on projectional editors (e.g., [12, 5]) uses a special graphical user interface to allow the developer to implicitly mark where the extensions are placed in the code, essentially directly specifying the underlying ASTs. This solution to the language extension problem poses several challenges, such as defining and implementing the semantics for the composition of the languages and the channels for communication between them. In Wyvern, we do not encounter these problems as the semantic rules for a language composition are incorporated within the host language by design.

Recent work on Active Code Completion (ACC) associates code completion palettes with types [24], much as we associate parsers with types. ACC palettes could be used for defining a TSL syntax for types. However, in ACC that syntax is immediately translated to Java syntax at edit time, while this work integrates with the core parsing facilities of the language.

## 8 Discussion

*Safe TSL Composition* Our primary contribution is a strategy for composing TSLs with each other and with a host language, that ensures that ambiguities cannot occur. The host language ensures that TSLs are delimited unambiguously, and the TSL ensures that the host language is delimited unambiguously. TSLs can be nested safely by briefly entering the host language. The body of the TSL is interpreted by a fixed grammar – the one associated with its expected type. This avoids the kinds of conflicts a simple merger of the grammars would cause. Apart from the large number of TSLs that can be composed together in a short piece of code while producing meaningful results, we

aim to provide a safe composability guarantee that other language extension solutions do not [7, 15].

*Keyword-Directed Invocation* In most domain-specific language frameworks, a switch to a different language is indicated by a keyword or function call naming the language to be used. Wyvern eliminates this overhead in many cases by determining the TSL based on the expected type of an expression. This lightweight mechanism is particularly useful for small languages. Keyword-directed invocation is simply a special case of our type-directed approach. In particular, a keyword macro can be defined as a function with a single argument of a type specific to that keyword. The type contains the implementation of the domain-specific syntax associated with that keyword. In the most general sense, it may simply allow the entire Wyvern grammar, manipulating it in later phases of compilation.

As an example, consider control flow operators like `if`. This can be defined as a polymorphic method of the `bool` type with signature $(\texttt{unit} \to \alpha, \texttt{unit} \to \alpha) \to \alpha$. That is, it takes the two branches as functions and chooses which to invoke based on the value of the boolean, using perhaps a more primitive control flow operator, like case analysis, or even a Church encoding of booleans as functions. In Wyvern, the branches could be packaged together into a type, `IfBranches`, with an associated grammar that accepts the two branches as unwrapped expressions. Thus, `if` could be defined entirely in a library and used as follows:

```
1  if(guard, ~)
2    then
3      <any Wyvern>
4    else
5      <any Wyvern>
```

For methods like `if` where constructing the argument explicitly will almost never be done, it may be useful to mark the method in a way that allows Wyvern to assume it is being called with a TSL argument immediately following its use. This would eliminate the need for the `(~)` portion, supporting even more conventional notation.

*Interaction with Subtyping* The mechanism described here does not consider the case where multiple subtypes of a base type define a grammar. This can be resolved in several ways. Our plan in full Wyvern, which includes subtyping, is to use the *declared* type's grammar (if a subtype's grammar is desired, an explicit type annotation on the tilde can be used). Alternatively, we could attempt to parse against all relevant subtypes, only requiring explicit disambiguation when ambiguities arise.

*Custom Lexers* Our existing lexing strategy may be too restrictive, requiring all DSLs to be hierarchical in nature. One potential expansion would be to enable DSLs to define their own lexers, still perhaps delimited by indentation or parentheses. Such an extension would sacrifice some readability.

We do not allow a replacement parser for infix operators as we considered it to unnecessarily complicate the current prefixed parsing approach. In the future, we plan to further support redefining operators.

## Acknowledgements

We thank the anonymous reviewers for helpful comments, and acknowledge the support of the United States Air Force Research Laboratory and the National Security Agency lablet contract #H98230-14-C-0140, as well as the Royal Society of New Zealand Marsden Fund. Cyrus Omar is supported by an NSF Graduate Research Fellowship.

## References

1. M. D. Adams. Principled Parsing for Indentation-Sensitive Languages: Revisiting Landin's Offside Rule. In *Principles of Programming Languages*, pages 511–522. ACM, 2013.
2. M. Bravenboer, R. Vermaas, J. Vinju, and E. Visser. Generalized type-based disambiguation of meta programs with concrete object syntax. In *Generative Programming and Component Engineering*, pages 157–172. Springer, 2005.
3. T. Clark, P. Sammut, and J. S. Willans. Beyond annotations: A proposal for extensible java (XJ). In *Source Code Analysis and Manipulation*, pages 229–238. IEEE, 2008.
4. D. de Rauglaudre. Camlp4 - Reference Manual. `http://caml.inria.fr/pub/docs/manual-camlp4/`, 2003.
5. L. Diekmann and L. Tratt. Parsing composed grammars with language boxes. In *Workshop on Scalable Language Specification*, 2013.
6. S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. SugarJ: library-based language extensibility. In *Object-Oriented Programming Systems, Languages, and Applications*, pages 391–406. ACM, 2011.
7. S. Erdweg and F. Rieger. A framework for extensible languages. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, pages 3–12. ACM, 2013.
8. T. Green and M. Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7(2):131–174, 1996.
9. R. Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2012.
10. R. Harper and C. Stone. A Type-Theoretic Interpretation of Standard ML. In *IN Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.
11. K. Ichikawa and S. Chiba. Composable user-defined operators that can express user-defined literals. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, pages 13–24, New York, NY, USA, 2014. ACM.
12. JetBrains. JetBrains MPS – Meta Programming System. `http://www.jetbrains.com/mps/`.
13. V. Karakoidas. On domain-specific languages usage (why dlss really matter). *XRDS*, 20(3):16–17, Mar. 2014.
14. L. C. L. Kats and E. Visser. The Spoofax Language Workbench. Rules for Declarative Specification of Languages and IDEs. In *Object-Oriented Programming Systems, Languages, and Applications*, 2010.
15. H. Krahn, B. Rumpe, and S. Völkel. Monticore: Modular development of textual domain specific languages. In *Objects, Components, Models and Patterns*, 2008.
16. L. Krishnan and E. V. Wyk. Termination analysis for higher-order attribute grammars. In *SLE*, pages 44–63, 2012.
17. B. Lee, R. Grimm, M. Hirzel, and K. S. McKinley. Marco: Safe, expressive macros for any language. In *ECOOP*, volume LNCS 7313, pages 356–382. Springer, 2012.
18. X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.
19. A. Loh, T. van der Storm, and W. R. Cook. Managed data: Modular strategies for data abstraction. In *Onward!*, pages 179–194. ACM, 2012.

20. Microsoft Corporation. Expression Trees (C# and Visual Basic). `http://msdn.microsoft.com/en-us/library/bb397951.aspx`.
21. E. Möller. SRFI-49: Indentation-sensitive syntax. `http://srfi.schemers.org/srfi-49/srfi-49.html`, 2005.
22. L. Nistor, D. Kurilova, S. Balzer, B. Chung, A. Potanin, and J. Aldrich. Wyvern: A simple, typed, and pure object-oriented language. In *Proceedings of the 5th Workshop on Mechanisms for SPEcialization, Generalization and inHerItance*, MASPEGHI '13, pages 9–16, New York, NY, USA, 2013. ACM.
23. C. Omar, B. Chung, D. Kurilova, A. Potanin, and J. Aldrich. Type-directed, whitespace-delimited parsing for embedded dsls. In *Proceedings of the First Workshop on the Globalization of Domain Specific Languages*, GlobalDSL '13, pages 8–11, New York, NY, USA, 2013. ACM.
24. C. Omar, Y. Yoon, T. D. LaToza, and B. A. Myers. Active code completion. In *International Conference on Software Engineering*, 2012.
25. OWASP. OWASP Top 10 2013. `https://www.owasp.org/index.php/Top_10_2013-Top_10`, 2013.
26. Z. Palmer and S. F. Smith. Backstage Java: Making a Difference in Metaprogramming. In *Object-Oriented Programming Systems, Languages, and Applications*, 2011.
27. B. C. Pierce and D. N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, Jan. 2000.
28. A. C. Schwerdfeger and E. R. Van Wyk. Verifiable composition of deterministic grammars. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 199–210, New York, NY, USA, 2009. ACM.
29. T. Sheard and S. Jones. Template meta-programming for haskell. *ACM SIGPLAN Notices*, 37(12):60–75, 2002.
30. J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA, USA, 1977.
31. M. Tatsubori, S. Chiba, M.-O. Killijian, and K. Itano. OpenJava: A Class-based Macro System for Java. In *Reflection and Software Engineering*, 2000.
32. E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. Qualitas corpus: A curated collection of java code for empirical studies. In *Proc. 2010 Asia Pacific Software Engineering Conference (APSEC'10)*, 2010.
33. L. Tratt. Domain specific language implementation via compile-time meta-programming. *ACM Trans. Program. Lang. Syst.*, 30(6):31:1–31:40, Oct. 2008.
34. M. G. J. van den Brand. *Pregmatic: A Generator for Incremental Programming Environments*. PhD thesis, Katholieke Universiteit Nijmegen, 1992.
35. E. R. Van Wyk and A. C. Schwerdfeger. Context-aware scanning for parsing extensible languages. In *Generative programming and component engineering*, pages 63–72. ACM, 2007.