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 syntax 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 syntax. Frustration with this causes developers to use strings, rather than structured data, with alarming frequency, leading to correctness, performance, security, and usability issues. Allowing library providers to modularly extend a language with new syntax could help address these issues. Unfortunately, prior mechanisms either limit expressiveness or are not safely composable: individually unambiguous extensions can still cause 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 programming language.

Keywords: extensible languages; parsing; bidirectional typechecking; hygiene

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 [31]:

> 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 [26]. 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 in 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. For clarity of presentation, we color each character by the TSL it is governed by. Black is the base language and comments are in italics.

```
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

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 (below). We could have created a value of type `URL` using the general-purpose introductory form **`new`**, which *forward references* an indented block of field and method definitions beginning on the line after it appears:

```
objtype URL
  val protocol : String
  val subdomain : String
  (* ... *)
```

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

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 indicated can be obeyed. The type annotation on `imageBase` (or equivalently, ascribed directly to 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) and produce an *elaboration*: a Wyvern abstract syntax tree (AST) that explicitly instantiates a new object of type `URL` using general-purpose forms only, 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 here delimited by percent signs, as seen on line 2 of Fig. 1. The TSL chooses to parse code between percent signs as a Wyvern expression, using its abstract syntax tree (AST) to construct the overall elaboration. A string-based representation of the URL is never constructed 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. `BodyElement((attrs, child))`. But, as discussed in the introduction, this can be cognitively demanding. Thus, we have associated 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, requiring awkward and error-prone escaping. It also avoids issues with leading indentation in multi-line literals, as the parser strips these automatically for layout-delimited literal bodies.

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 using a general mechanism for associating a statically-known value with a named type, called its *metadata*. Type metadata, in this context, is comparable to class annotations in Java or class/type 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. Note that we allow programs to extract the metadata value of a named type `T` programmatically using the form **`metadata`**`[T]`.

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 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. 5 shows portions of the decla-

```
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) => Inj(`BodyElement`, Pair(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
  Inj of Id * Exp
  ...
  Spliced of ParseStream
  metadata : HasTSL = new
    val parser = (*quasiquotes*)
```

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

rations 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 specialized notation for defining a parser, so we can implement a grammar-based parser generator as a TSL atop the lower-level interface exposed by `Parser`. We do so using a layout-sensitive grammar formalism developed by Adams [1]. Wyvern is itself layout-sensitive and has a grammar that can be written down using this formalism, as we will discuss, so it is sensible to expose it to TSL providers as well. Most aspects of this formalism are conventional. Each non-terminal (e.g. the designated `start` non-terminal) is defined by a number of disjunctive rules, each introduced using `<-`. Each rule 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 rule 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). Note that the leftmost column is not simply the first character, in the case of terms that span multiple lines. 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 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, though not always, 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 Adams grammars, the syntax of real-world languages like Python and Haskell can be written declaratively.

Each rule is followed, in an indented block, by a spliced function that generates an elaboration given the elaborations recursively generated by each of the n non-terminals in the rule, ordered left-to-right. Elaborations are of type `Exp`, which is a case type containing each form in the abstract syntax of Wyvern (as well as an additional case, `Spliced`, that is used internally), which we will describe later. Here, we show how to generate an elaboration using the general-purpose introductory form for case types (line 11, `Inj` corresponds to the introductory form for case types) as well as using *quasiquotes* (line 13). Quasiquotes are expressions written in concrete syntax that are not evaluated for their value, but rather evaluate to their corresponding 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. 5). They support the 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, not string-based.

We can see how HTML splicing works on lines 12-15: we simply include the Wyvern expression non-terminal `EXP` in our rule and insert it into our quoted result where appropriate. The type that the spliced Wyvern expression will be expected to have is determined by where it is placed. 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.

3 Syntax

3.1 Concrete Syntax

We will begin our formal treatment by specifying the concrete syntax of Wyvern declaratively, using the same layout-sensitive formalism that we have introduced for TSL grammars, developed recently by Adams [1]. Adams grammars are useful because they allow 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 could not 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 the quasiquote TSL for `Exp` more easily.

3.2 Program Structure

The concrete syntax of TSL Wyvern is shown in Fig 6. An example Wyvern program showing several unique syntactic features of TSL Wyvern is shown in Fig. 7 (left).

```
(* 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= '('> ã̄l[fwd]> ')'>
ẽ[fwd] → '('> ã̄l[fwd]> ')'>
ẽ[fwd] → ẽ[fwd]= '.'> ID>
ẽ[fwd] → 'toast'= '('> ẽ[fwd]> ')'>
ã̄l[fwd] → ẽ[fwd]= | ẽ[fwd]= ','> ā̄l_nonempty> | ē= ','> ã̄l[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.

```
objtype T
  val y : HTML
let page : HTML->HTML = (fn(x) => ~)
  >html
    >body
      <{x}
page(case(5 : Nat))
  Z(_) => (new : T).y
    val y = ~
      >h1 Zero!
  S(x) => ~
    >h1 Successor!
```

$$
\begin{array}{l}
\textbf{objtype}[T, (y[\textbf{named}[HTML]], \emptyset), ()]; \emptyset; \\
\textbf{elet}(\textbf{easc}[\textbf{arrow}[\textbf{named}[HTML], \\
\quad \textbf{named}[HTML]]](\textbf{elam}(x.\textbf{lit}[>\textit{html} \\
\quad >\textit{body} \\
\quad\quad <\{x\}])), page. \\
\textbf{eap}(page; \textbf{ecase}(\textbf{easc}[\textbf{named}[Nat]](\textbf{lit}[5])) \{ \\
\quad \textbf{erule}[Z](\_.\textbf{eprj}[y](\textbf{easc}[\textbf{named}[T]](\textbf{enew}\{ \\
\quad\quad \textbf{eval}[y](\textbf{lit}[>\textit{h1 Zero!}]); \emptyset\}))); \\
\quad \textbf{erule}[S](x.\textbf{lit}[>\textit{h1 Sucessor!}]); \emptyset \\
\}))
\end{array}
$$

Fig. 7: An example Wyvern program demonstrating all three forward referenced forms. The corresponding abstract syntax is on the right.

The top level of a program (the `p` non-terminal) consists of a series of named type declarations – object types using **`objtype`** or case types using **`casetype`** – followed by an expression, `e`. Each named type declaration 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 metadata). In the grammar, sequences of top-level declarations use the form $\texttt{p}^{=}$ to signify that all the succeeding `p` terms must begin at the same indentation. We do not specify separate compilation here, as this is an orthogonal issue.

3.3 Forward Referenced Blocks

Wyvern makes extensive use of forward referenced blocks to make its syntax clean. In particular, layout-delimited TSLs, **`new`** expressions for introducing objects, and **`case`** expressions for eliminating case types and tuples all make use of forward referenced blocks. Fig. 7 shows these in use (assuming suitable definitions of `Nat` and `HTML`).

Each line in the concrete syntax can contain either zero or one forward references. We distinguish these in the grammar by defining separate non-terminals $\overline{\texttt{e}}$ and $\widetilde{\texttt{e}}\texttt{[fwd]}$, where the parameter `fwd` is the particular forward reference form that occurs. Note particularly the rule for **`let`** (which permits an expression to span multiple lines and so can be used to support multiple forward references in a single expression).

3.4 Abstract Syntax

The concrete syntax of a Wyvern program, `p`, is parsed to a program in the abstract syntax, ρ, shown in Fig. 8. Forward references are internalized. Note that all literal forms are unified into the abstract literal form $\textbf{lit}[body]$, including the layout-delimited form and number literals. The body remains completely unparsed at this stage. The abstract syntax for the example in Fig. 7 is shown to its right and demonstrates the key rewriting done at this stage. Simple product types can be rewritten as object types in this phase. We assume that this occurs so that we can avoid specifying them separately in the remainder of the paper, though we continue to use tuple notation for concision.

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, Γ, maps bound variables to types, and the named type context, Θ, maps

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

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

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

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

Fig. 8: Abstract Syntax of TSL Wyvern programs (ρ), type declarations (θ), types (τ), external terms (e), translational terms ($\hat{e}$) and internal terms (i) and auxiliary forms. Metavariable T ranges over type names, ℓ 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 unit as $()$ and pairs as (i_1, i_2) in abstract syntax as needed.

type names to their declarations. Such typing judgements do not fully specify whether, when writing a typechecker, the type should be considered an input or an output. In some situations, a type propagates in 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. In others, 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* [28] make this distinction explicit by specifying the type system instead using two simultaneously defined typechecking judgements corresponding to these two situations.

To support TSLs, we need to also, simultaneously with this process, perform an elaboration from external terms, which contain literals, to *internal terms*, i, the syntax for which is shown on the right side of Fig. 8. Internal terms contain neither literals nor the 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). This manner of specifying a type-directed mapping from external terms to a smaller collection of internal terms, which are the only terms that are given a dynamic

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

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

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

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

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

$\boxed{\vdash_\Theta \omega}$ $\dfrac{\ell \notin \mathrm{dom}(\omega) \quad \vdash_\Theta \tau \quad \vdash_\Theta \omega}{\vdash_\Theta \ell[\tau]; \omega}$ *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, ρ. Note that type declarations can only be recursive, not mutually recursive, with these rules. The prelude Θ_0 (see Fig. 5) defines mutually recursive types, so we cannot write a θ_0 corresponding to Θ_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 [24].

semantics, is related to the Harper-Stone elaboration semantics for Standard ML [10]. Note that both terms share a type system.

Our static semantics are thus formulated by combining these two ideas, forming a *bidirectionally typed elaboration semantics*. The judgement $\Gamma \vdash_\Theta e \rightsquigarrow i \Rightarrow \tau$ means that under typing context Γ and named type context Θ, external term e elaborates to internal term i and synthesizes type τ. The judgement $\Gamma \vdash_\Theta e \rightsquigarrow i \Leftarrow \tau$ is analagous but for situations where we are analyzing e against type τ.

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, Θ_0 (see Fig. 5), then validate the provided user-defined type declarations, θ, to produce a corresponding named typed context, Θ. 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 Θ (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 τ and produce 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 $\Theta_0 \Theta$.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.2 External Terms

The bidirectional typechecking and elaboration rules for external terms are specified beginning in Fig. 10. Most of the rules are standard for a simply typed lambda calculus with labeled sums and labeled products, and the elaborations are direct to a corresponding internal form. 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, as in many simple formulations, all introductory forms are analytic and all elimination forms are synthetic, though this can be relaxed in practice to support some additional idioms.

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 ω 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, χ. 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 χ appears in a rule in the rule sequence r, elaborating it to the internal rule sequence $\dot{r}$. Every rule must synthesize the same type, τ.

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 specified by 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 in the failure case to report the error to the user.

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.

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

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

Fig. 11: Statics for external terms, e, continued. This is the key rule (described below).

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 perform this check explicitly (as an additional premise).
3. A parse stream, i_{ps}, which is an internal term of type $\mathbf{named}[ParseStream]$, is generated from the body of the literal. This is an object that allows the TSL to read the body and supports some additional conveniences, discussed further below.
4. The `parse` method is called with this parse stream. If it produces the appropriate case containing a *reified elaboration*, i_{ast} (of type `Exp`) and the 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 (the maximal transitive closure, if it exists, of the small-step evaluation relation in 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 $\mathbf{spliced}[e]$, 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. The reified form, unlike the translational form it corresponds to, does not contain the expression itself, but rather just the portion of the parse stream that should be treated as spliced. Because parse streams (and thus portions thereof) can originate only metatheoretically (i.e. from the compiler), 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, below.

   The convenience 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 Γ_{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 is hygienic, as we will describe next. Each rule in Fig. 10 should be thought of as having a corresponding rule in Fig. 14. Two examples are shown for concision.

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

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

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

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

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

$$\cdots$$

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

$\boxed{i \uparrow \tau}$

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

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

Fig. 12: Dereification rules, used by rule *T-lit* (above) to determine the translational term encoded by the internal term of type **named**$[Exp]$. We assume a bijection between internal terms of type **named**$[ID]$ (written i_{id}) and variables, type names and case and member labels.

$\boxed{i \downarrow i}$

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

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

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

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

$$\cdots$$

$\boxed{\tau \downarrow i}$

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

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

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).

4.4 Hygiene

A concern with any term rewriting system is *hygiene* – how should variables in the elaboration 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 then, given our setting, prevent even spliced terms from referring to local variables. These are written by the TSL client, who is aware of variable bindings at the use site, so this should be permitted.

Furthermore, the variables in spliced terms should be bound as 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. by introducing a variable `tmp` outside a spliced term that "leaks" into the spliced term).

The solution to both of these issues, given what we have outlined above, is now quite simple: we have constructed the system so that we know which sub-terms originate from the TSL client, marking them as **spliced**$[e]$. These terms are permitted to refer only to

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

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

$$\cdots$$

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

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. Note that elaboration is implicitly capture-avoiding here (see Sec. 6).

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

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.

variables in the client's context, Γ_{out}, as seen in the premises of the two rules pertaining to this form (one for analysis, one for synthesis). The portions of the elaboration that aren't marked in this way were generated by the TSL provider, so they can refer only to variables introduced earlier in the elaboration, tracked by the context Γ, initially empty. The two are kept separate. If the TSL wishes to introduce values into 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

By this formulation, elaborations containing free variables are always erroneous. In some rewriting systems, a free variable is not an error, but are instead replaced with the AST corresponding to the value of the variable at the generation site. We permit this explicitly by including the form `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. The rules for reification, used here, and dereification, used in the literal rule above, are dual.

The TSL associated with `Exp`, implementing quasiquotes, can perform free variable analysis and insert this form automatically, so they need not be inserted manually in most cases. That is, `Var('x') : Exp` elaborates to x which is ill-typed in an empty context, `'x' : Exp` produces the translational term **htoast**(**spliced**$[x]$), which will elaborate to **itoast**(x) in the context where the quotation appears (i.e. in the TSL definition), thus behaving as described without requiring that quotations are entirely implemented by the language. This can be seen as a form of serialization and could be implemented as a library using reflection or compile-time metaprogramming techniques (e.g. [20]).

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

Fig. 16: Statics for internal terms, i. Each rule in Fig. 10 except *T-metadata* 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 Metatheory

The semantics we have defined constitute a type safe language. We will outline the key theorems and lemmas here, referring the reader to an accompanying technical report for fuller details [24]. The two key theorems are: internal type safety, and type preservation of the elaboration process.

To prove internal type safety, we must define a bidirectional typing judgement for the internal language, shown and described in Fig. 16 (by the external type preservation theorem, we should never need to explicitly implement this, however). We must also define a well-formedness judgement for named type contexts (not shown).

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 **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* $\emptyset \vdash_\Theta i' \Leftarrow$ ***named***$[Exp]$*.*

Proof. The proof is by a straightforward induction. Analagous 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 to open terms by defining a well-formedness judgement for contexts (not shown). 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 derivation. Nearly all the elaborations are direct, so the proof is by straightforward applications of induction hypotheses and lemmas about well-formed contexts. 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 Θ implies that the metadata term is of the type assigned.

– $e = \textbf{lit}[body]$. 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 due to alpha renaming) 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 by induction over the typing derivation follows the same outline as above for all the shared cases. The outer context is threaded through opaquely when applying the inductive hypothesis. The only rules of note are the two 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. □

Moving up to the level of programs, we can prove the correctness of compilation theorem below. Together, this implies that derivation of the compilation judgement produces an internal term that does not go wrong.

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

Proof. We simply need a lemma about checking type declarations and the result follows straightforwardly.

Lemma 3 (Type Declaration). *If* $\vdash_{\Theta_0} \theta \sim \Theta$ *then* $\vdash \Theta_0\Theta$.

Proof. The proof is a simple induction using the definition of $\vdash \Theta$ (not shown).

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, because TSL Wyvern is not a total language (due to self-reference in methods). Indecidability of typechecking is strictly for this reason. Typechecking of terms not containing literals is guaranteed to terminate. Termination of parsers and parser generators has previously been studied (e.g. [15]) 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 [17]), have made it a priority to fully verify and prove termination of the parser, because it is perceived that most bugs in compilers arise due to incorrect optimization passes, not initial parsing.

5 Corpus Analysis

We performed a corpus analysis on existing Java code to assess how frequently there are opportunities to use TSLs. As a lower bound for this metric, 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 instantiating an object.

Methodology. We ran our analysis on a recent version (20130901r) of the Qualitas Corpus [33], 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. Some additional details are provided in the technical report [24].

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.

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

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%**

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, assignments, etc., that could possibly use TSLs. We did not count types that themselves could have a TSL associated with them (e.g. `URL`), only uses of `String`s that we hypothesized might not have been `String`s had better syntax been available. Our search for constructors with the use of command line tools and text editor features may not have identified every 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 are many potential use cases where type-specific languages could be considered, given that numerous `String` arguments appeared to specify a parseable format.

6 Implementation

Because Wyvern itself is an evolving language and we believe that the techniques herein are broadly applicable, we have implemented the abstract syntax, typechecking and elaboration rules precisely as specified in this paper, including the hygiene mechanism, in Scala as a stable resource. We have also included a simple compiler from our representation of internal terms, which includes explicit type information at each node, to Scala source code. We represent both external terms and translational terms using the same case classes, using traits to distinguish them when necessary. This code can be used to better understand the implementation overhead of our mechanisms. The key "trick" is to make sure that the typing context also maps each source variable to a unique internal variable, so that elaboration of spliced terms is capture-avoiding. This code can be found at `http://github.com/wyvernlang/tslwyvern`.

Wyvern itself also supports a variant of this mechanism. The Wyvern language is an evolving effort involving a number of techniques other than TSLs, so the implementation does not precisely coincide with the specification presented herein. In particular, Wyvern's object types and case types have substantially different semantics. Moreover, Adams grammars do not presently have a robust implementation, so their presentation here is merely expository. The top-level parser for Wyvern is instead produced by the Copper parser generator [36] which uses stateful LALR parsing to handle whitespace. Forward references, such as the TSL tilde, the new keyword, and case expressions, are handled by inserting a special "signal" token into the parse stream at the end of an expression containing a forward reference. When the parser subsequently reads this signal token, it enters the appropriate state depending on the type of forward reference encountered. TSL blocks are handled as if they were strings, preserving all non-leading whitespace, and new and case expression bodies are parsed using their respective grammars. Wyvern performs literal parsing during typechecking essentially as described, using a standard bidirectional type system. It does not enforce the constraints on parse streams and the hygiene mechanisms as of this writing. Some of the API is implemented using a Java interoperability layer rather than directly in Wyvern. This implementation does support some simpler examples fully, however (unlike the implementation above, which does not have a concrete syntax at all). The code can be found at `http://github.com/wyvernlang/wyvern`.

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 annotated with named types. 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 ProteaJ allows the programmer to explicitly disambiguate, as in other systems. In contrast, by associating parsers with types, our approach avoids all conflicts, achieving a stricter notion of modularity at the cost of some expressiveness (we only consider delimited literals – these may define operators inside, but we cannot support custom operator syntax directly at the top level). We also give a type theoretic foundation for our approach.

Another way to approach language extensibility is to go a level of abstraction above parsing, as is done via metaprogramming and macro facilities, with Scheme and other Lisp-style languages' hygienic macros being the 'gold standard' for hygiene. In those languages, macros are written in the language itself and use its 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 maintain a static type discipline. 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. Static macro systems also exist. For instance, OJ (previously, OpenJava) [32] provides a macro system based on a meta-object protocol, and Backstage Java [27], Template Haskell [30] and Converge [34] also employ compile-time meta-programming, the latter with some support for whitespace delimited blocks. Each of these systems provide macro-style rewriting of source code, but they provide at most limited extension of language parsing. String literals can be reinterpreted, but splicing is not hygienic if this is done.

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 [19] are similar in that, when the type of a term is, e.g., `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 as one use case amongst many.

Many approaches to syntax extension, such as XJ [3] are keyword-delimited in some form. We believe that a type-directed approach is more seamless and natural, coinciding with how one would build in language support directly. These approaches also differ in that they either do not support hygienic expansion, or have not specified it in the simple manner that we have.

In terms of work on safe language composition, Schwerdfeger and van Wyk [29] 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, as with 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 [18] and others [35], 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 [16] 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 *internal* extensibility, 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 is of considerable interest to us, but remains relatively understudied formally. It is likely that a type-oriented approach to projectional editing, inspired by that described herein, could be fruitful.

We were informed by our previous work on Active Code Completion (ACC), which associates code completion palettes with types [25], much as we associate parsers with types. ACC palettes could be used for defining a TSL syntax for types in a complementary manner. In ACC that syntax is immediately translated to Java syntax at edit time, while this work integrates with the language, so the syntax is retained with the code. ACC supports more general interaction modes than just textual syntax, situated between our approach and projectional editors.

8 Discussion

We have presented a minimal but complete language design that we believe is particularly elegant, practical and theoretically well-motivated. The key to this is our organization of language extensions around types, rather than around grammar fragments.

There are several directions that remain to be explored:

- TSL Wyvern does not support polymorphic types, like `'a list` in our first example. Were we to add support for them, we would expect that the type constructor (`list`) would determine the syntax, not the particular type. Thus, we may fundamentally be proposing *type constructor specific languages*.
- Similarly, TSL Wyvern does not support abstract types. It may be useful to include the ability to associate metadata with an abstract type, much in the same way that we associate metadata with a named type here.
- TSLs as described here allow one to give an alternative syntax for introductory term forms, but elimination forms cannot be defined directly. There are two directions we may wish to go to support this:
  1. Pattern matching is a powerful feature supported by an increasing number of languages. Pattern syntax is similar to term syntax. It may be possible for a TSL definition to include parse functions for "literal-like" forms appearing in patterns, elaborating them to pattern terms rather than expression terms.
  2. Keywords are more useful when defining custom elimination forms (e.g. `if` based on `case`). It may be possible to support "typed syntax macros" using the same hygiene mechanisms we described here.
- We do not provide TSLs with the ability to diverge based on the type of a spliced expression. This might be useful if, for example, our HTML TSL wanted to treat spliced strings differently from other spliced HTML terms. For polymorphic types, we might also wish to diverge based on the type index.

- We may wish to design less restrictive shadowing constraints, so that TSLs can introduce variables directly into the scope of a spliced expression if they explicitly wish to (bypassing the need for the client to provide a function for the TSL to call). The community may wish to discuss whether this is worth the cost in terms of difficulty of determining where a variable has been bound.
- We need to provide further empirical validation. This may benefit from the integration of TSLs into existing languages other than Wyvern.
- We need to consider broader IDE support – custom syntax benefits from custom editor support, and it may be possible to design IDEs that dispatch to type metadata in much the way the typechecker does in this paper. Our informal considerations of existing IDE extension mechanisms suggests that this may be non-trivial.

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 was 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*, 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*, 2005.
3. T. Clark, P. Sammut, and J. S. Willans. Beyond annotations: A proposal for extensible Java (XJ). In *Source Code Analysis and Manipulation*, 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*, 2011.
7. S. Erdweg and F. Rieger. A framework for extensible languages. In *Generative Programming: Concepts & Experiences*, 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 *Modularity*, 2014.
12. JetBrains. JetBrains MPS – Meta Programming System. `http://www.jetbrains.com/mps/`.
13. V. Karakoidas. On domain-specific languages usage (why DSLs really matter). *Crossroads*, 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. L. Krishnan and E. Van Wyk. Termination analysis for higher-order attribute grammars. In *Software Language Engineering*, 2012.
16. B. Lee, R. Grimm, M. Hirzel, and K. S. McKinley. Marco: Safe, expressive macros for any language. In *European Conference on Object-Oriented Programming*, 2012.
17. X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.
18. A. Loh, T. van der Storm, and W. R. Cook. Managed data: Modular strategies for data abstraction. In *Onward!*, 2012.
19. Microsoft Corporation. Expression Trees (C# and Visual Basic). `http://msdn.microsoft.com/en-us/library/bb397951.aspx`.
20. H. Miller, P. Haller, E. Burmako, and M. Odersky. Instant pickles: Generating object-oriented pickler combinators for fast and extensible serialization. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 183–202, New York, NY, USA, 2013. ACM.
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 *MechAnisms for SPEcialization, Generalization and inHerItance*, 2013.
23. C. Omar, B. Chung, D. Kurilova, A. Potanin, and J. Aldrich. Type-directed, whitespace-delimited parsing for embedded DSLs. In *Globalization of Domain Specific Languages*, 2013.
24. C. Omar, D. Kurilova, L. Nistor, B. Chung, A. Potanin, and J. Aldrich. Safely composable type-specific languages. Carnegie Mellon University technical report CMU-ISR-14-106, 2014.
25. C. Omar, Y. Yoon, T. D. LaToza, and B. A. Myers. Active code completion. In *International Conference on Software Engineering*, 2012.
26. OWASP. OWASP Top 10 2013. `https://www.owasp.org/index.php/Top_10_2013-Top_10`, 2013.
27. Z. Palmer and S. F. Smith. Backstage Java: Making a Difference in Metaprogramming. In *Object-Oriented Programming Systems, Languages, and Applications*, 2011.
28. B. C. Pierce and D. N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, Jan. 2000.
29. A. C. Schwerdfeger and E. R. Van Wyk. Verifiable composition of deterministic grammars. In *Programming Language Design and Implementation*, 2009.
30. T. Sheard and S. Jones. Template meta-programming for Haskell. *ACM SIGPLAN Notices*, 37(12):60–75, 2002.
31. J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA, USA, 1977.
32. M. Tatsubori, S. Chiba, M.-O. Killijian, and K. Itano. OpenJava: A Class-based Macro System for Java. In *Reflection and Software Engineering*, 2000.
33. 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 *Asia Pacific Software Engineering Conference*, 2010.
34. L. Tratt. Domain specific language implementation via compile-time meta-programming. *ACM Trans. Program. Lang. Syst.*, 30(6), Oct. 2008.
35. M. G. J. van den Brand. *Pregmatic: A Generator for Incremental Programming Environments*. PhD thesis, Katholieke Universiteit Nijmegen, 1992.
36. E. R. Van Wyk and A. C. Schwerdfeger. Context-aware scanning for parsing extensible languages. In *Generative programming and component engineering*, 2007.