Safely Composable Type-Specific Languages

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 developers 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 cause ambiguities when used together. We introduce *type-specific languages* (TSLs): logic associated with a type that determines how *generic literal forms*, able to contain arbitrary syntax, are parsed and expanded, hygienically, into general-purpose syntax. 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 specify it with a bidirectional type system for an emerging language, Wyvern.

Keywords: extensible languages; parsing; bidirectional typechecking

1 Motivation

By using a general-purpose abstraction mechanism to encode a data structure, one immediately benefits from a body of established reasoning principles and primitive operations. For example, inductive datatypes can be used to express data structures like lists: intuitively, a list can either be empty, or be broken down into a value (its *head*) and another list (its *tail*). In an ML-like language, this concept is conventionally written:

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

By encoding lists in this way, we can reason about them by structural induction, construct them by choosing the appropriate case and inspect them by pattern matching. In object-oriented languages, one can encode lists similarly as singly-linked cells, reason about them using a variety of program analysis techniques, construct them using `new` and inspect them by traversing the links iteratively. In each case, the programmer only needs to provide an encoding of the structure of lists; the semantics are inherited.

While inheriting semantics can be quite useful, inheriting associated general-purpose syntax can sometimes be a liability. 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))))
```

General-purpose object-oriented notation is similarly inconvenient:

```
new Cons<int>(1, new Cons<int>(2, new Cons<int>(3, new Cons<int>(4))))
```

Because lists are a common data structure, many languages provide specialized notation for constructing them, e.g. `[1, 2, 3, 4]`. This notation is semantically equivalent to the general-purpose notation shown above, but brings cognitive benefits by drawing attention to the content of the list, rather than the nature of the encoding. More specifically, it is more *terse*, *visible* and *maps more closely* to the intuitive notion of a list, to use terminology from the literature on the cognitive dimensions of notations [13].

Although number, string and list literals are nearly ubiquitous features of modern languages, some languages also provide specialized notation for other common data structures, like maps and sets, data formats, like XML and JSON, query languages, like regular expressions and SQL, and markup languages, like HTML. For example, a language with built-in syntax for HTML and SQL, with type-safe interpolation of host language terms via curly braces, might allow:

```
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>
```

to mean:

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

When a specialized notation is not available, and equivalent general-purpose notation is too cognitively demanding for comfort, developers typically turn to run-time mechanisms to make constructing data structures more convenient. Among the most common strategies in these situations, as we will discuss in Sec. 5, is to simply use a string representation that is parsed at run-time. Developers are frequently tempted to write the example above as:

```
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 much of the notational convenience of the literal version, it is still more awkward to write, requiring explicit conversions to and from structured representations and escaping when the syntax of the language clashes with the syntax of string literals (line 2). But code like this also causes a number of more serious problems beyond cognitive load. Because parsing occurs at run-time, syntax errors will not be discovered statically, causing potential problems 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, or in a loop). But the most serious issue with this code is that it is fundamentally insecure: it is vulnerable to cross-site scripting attacks (line 1)

and SQL injection attacks (line 3). For example, if a user provided 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 [3]. Although developers are cautioned to sanitize their input, it can be difficult to verify that this was done correctly throughout a codebase. The most straightforward way to avoid these problems is to use structured representations throughout the codebase, aided by specialized notation like that above [5].

As we will discuss further in Sec. 5, situations like this, where specialized notation is necessary to maintain strong correctness, performance and security guarantees while avoiding unacceptable cognitive overhead, are quite common. Today, implementing new notations within an existing programming language requires the cooperation of the language designer. The primary technical reason for this is that, with conventional parsing strategies, not all notations can safely coexist, so a designer is needed to make choices about which syntactic forms are available and what their semantics are. For example, conventional notations for sets and dictionaries are both delimited by curly braces. When Python introduced set literals, it chose to distinguish them based on whether the literal contained only values, or key-value pairs. But this causes an ambiguity with the syntactic form `{ }` – should it mean an empty set or an empty dictionary? The designers of Python chose the latter interpretation for backwards compatibility.

Languages that allow users to introduce new syntax from within libraries hold promise, but because there is no longer a designer making decisions about such ambiguities, the burden of resolving them falls to the users of extensions. For example, SugarJ [11] and other extensible languages generated by Sugar* [12] allow users to extend the base syntax of the host language (e.g. Java) with new forms, like set and dictionary literals. New forms are imported transitively throughout a program. To resolve syntactic ambiguities that arise, users 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 an understanding of the underlying parser technology (in Sugar*, GLR parsing using SDF) and increases the cognitive load of using the conflicting notations (e.g. both sets and dictionaries) in the same file. These kinds of conflicts occur in a variety of circumstances: HTML and XML, different variants of SQL, JSON literals and dictionaries, or simply different implementations ("desugarings") of the same specialized syntax (e.g. two regular expression engines) cause problems.

In this paper, we describe an alternative parsing strategy that avoids these problems by shifting responsibility for parsing certain *generic literal forms* into the typechecker. The typechecker, in turn, defers responsibility to user-defined types, 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 rewrites this term to use only general-purpose notation. TSLs can contain expressions in the host language. This strategy avoids the problem of ambiguous syntax, because neither the base language nor TSLs are ever extended directly. It also avoids ambiguous semantics – the meaning of a form like `{ }` can differ depending on its type, so it is safe to use it for empty sets, dictionaries and other data

structures, like JSON literals. This also frees notation from being tied to the variant of a data structure built into the standard library, which sometimes does not provide the exact semantics that a programmer needs (for example, Python dictionaries do not preserve order, while JSON does):

```
let empty_set : Set = { }
let empty_dict : Dict = { }
let empty_json : JSON = { }
```

We develop our work as a variant of an emerging programming language called Wyvern [21]. To allow us to focus on the essence of our proposal, the variant of Wyvern we develop here is simpler than the variant previously described: it is purely functional (there are no effects or mutable state) and it does not enforce a uniform access principle for objects (fields can be accessed directly). It also adds inductive datatypes, 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 extends and makes concrete a mechanism sketched out in an earlier short workshop paper [22]. We make the following novel contributions:

1. We specify a more complete layout-sensitive concrete syntax and show how it can be written without the need for a context-sensitive lexer or parser. It now includes a variety of inline literals, provides a full specification for the whitespace-delimited literal form introduced by a forward reference, ~, and provides other forms of forward-referenced blocks.
2. We develop a general mechanism for associating metadata with a type. A TSL is then implemented by associating a parser (of type `Parser`) with a type. The parser is responsible for rewriting tokenstreams (of type `Tokenstream`) into Wyvern ASTs (of type `Exp`). These types are defined in the standard library.
3. This lower-level mechanism is general, but writing a hand-written parser and manipulating syntax trees manually is cognitively demanding. We observe that *grammars* and *quasiquotes* can both be seen as TSLs for parsers and ASTs respectively and discuss how to implement them as such.
4. A naïve rewriting strategy would be *unhygienic* – it could allow for the inadvertent capture of local variables. We show a novel mechanism that ensures hygiene by requiring that the generated AST is closed except for subtrees derived from portions of the user's tokenstream that are interpreted as nested Wyvern expressions. We also show how to explicitly refer to local values available in the parser definition (e.g. helper functions) in a safe way.
5. We formalize the static semantics and literal parsing rules of TSL Wyvern as a bidirectional type system. By distinguishing locations where an expression synthesizes a type from locations where an expression is being analyzed against a previously synthesized type, we can precisely state where generic literals can appear. This also formalizes the hygiene mechanism.
6. We provide several examples of TSLs throughout the paper, but to examine how broadly applicable the technique is, we conduct a simple corpus analysis, finding that string languages are used ubiquitously.

```
1  let imageBase : URL = <images.example.com>
2  let bgImage : URL = <%imageBase%/background.png>
3  new : SearchServer
4    def serve_results(searchQuery : String, page : Nat) : Unit =
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: %‘#aabbcc‘.darken(20pct)% }
12         :body
13           :h1 Results for {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
19           }
```

Fig. 1: Wyvern Example with Multiple TSLs

```
1  ‘literal body here, ‘‘inner backticks‘‘ must be doubled‘
2  ’literal body here, ’’inner single quotes’’ must be doubled’
3  "literal body here, ""inner double quotes"" must be doubled"
4  {literal body here, {inner braces} must be balanced}
5  [literal body here, [inner brackets] must be balanced]
6  <literal body here, <inner angle brackets> must be balanced>
7  12xyz (* no delimiters necessary for number literals *)
```

Fig. 2: Inline Generic Literal Forms

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.

2.1 Inline Literals

Our first TSL appears on line 1. The type of `imageBase` is `URL`, an *object type* containing several fields representing the components of a URL: its protocol, domain name, port, path and so on. Instead of creating a new object and setting these fields explicitly, however, we instantiate them using the *generic literal* `<images.example.com>`, using conventional notation for URLs. Because of the type annotation on `imageBase`, this literal's *expected type* is known to be `URL`, so the body of this literal (the characters between the angle brackets) is governed by the `URL` TSL. This TSL will parse it *at compile-time* to produce a Wyvern AST that explicitly instantiates a new object of type `URL` (see below). Any other delimited form in Fig. 2 could have been used equivalently.

In addition to defining conventional notation for URLs, this TSL supports *typed interpolation* of a `URL` expression to form a larger URL. The interpolated term is delimited by percent signs, as seen on line 2. The TSL parses the portion between percent signs as a Wyvern expression and specifies its expected type as `URL`. The TSL can then assume the interpolated value is of this type to construct an AST for the overall value. Untyped—and unsafe—interpolation of strings

```
1  casetype HTML
2      Empty of Unit
3    | Text of String
4    | Seq of HTML * HTML
5    | BodyElement of Attributes * HTML
6    | StyleElement  of Attributes * CSS
7    | ...
8    metadata = new : HasParser
9      val parser : Parser = ~
10       start -> ':body'= child::start>
11         `HTML.BodyElement(([], %child%)`
12       start -> ':style'= e::EXP[NEWLINE]>
13         let empty_attrs : Attributes = []
14         Exp.CaseIntro(Type.Var("HTML"), "StyleElement",
15           Exp.ProdIntro(valAST(empty_attr), e))
16       start -> '{'= e::EXP['}']>
17         `%e% : HTML`
```

Fig. 3: A Wyvern case type with an associated TSL.

does not occur. Note that the delimiters that can be used to go from Wyvern to a TSL are determined by Wyvern (those in Fig. 2) while the delimiters that can be used to go from a TSL back to Wyvern are chosen by the TSL (see Sec. 2.5).

2.2 General-Purpose Notation for Object Types

The general-purpose introductory form for object types is **`new`**. This form is a syntactic *forward reference* to the layout-delimited block of definitions beginning on the line immediately after the line **`new`** appears in (line 4 in this case), and ending when the indentation level has returned to the baseline, or when the file ends (after line 19 in this case). An object in TSL Wyvern can contain methods, introduced using **`def`**, and fields, introduced using **`val`**. Here we have just a single method, `serve_results` taking two arguments. Object types in TSL Wyvern are simple structural interfaces that constrain the signatures of fields and methods. The *type ascription* around **`new`** checks that the object being introduced satisfies the signature of `SearchServer` (not shown).

2.3 Layout-Delimited Literals

On line 5 of Fig. 1, we see a call to the function `serve`, 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. Declarations of some of these cases can be seen on lines 2-6 of Fig. 3. The general-purpose introductory form for a case type like `HTML` is, e.g., `HTML.BodyElement((attrs, child))`. But, as discussed in Sec. 1, using this syntax can be cognitively demanding. Thus, we associate a TSL with `HTML` that provides a simplified notation for writing HTML, shown being used on lines 6-20 of Fig. 1. This literal body is layout-delimited, rather than delimited by explicit tokens as in Fig. 2, and introduced by another 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 a general-purpose term of type `HTML` generated from the literal body by the TSL during compilation.

2.4 Implementing a TSL

Portions of the implementation of the TSL for `HTML` are shown on lines 8-17 of Fig. 3, written as a declarative grammar. A grammar is simply a specialized notation for generating a parser, and so this notation is itself implemented as a TSL. We will discuss the underlying implementation in the next section. A `Parser` is associated with a type using **`metadata`**, as shown. The **`metadata`** of a type `t` is simply a value that is associated with the type at both compile-time and run-time. It can be accessed using the syntactic form `t.`**`metadata`**.

In this case, we are basing our grammar formalism on the layout-sensitive formalism developed by Adams [4]. Most aspects of this formalism are completely conventional. Each non-terminal (e.g. `start`) is defined by a number of disjunctive productions. Each production defines a sequence of terminals (e.g. `':body'`) and non-terminals (e.g. `start`), which can have names (e.g `child`) associated with them using `::`. In Adams' formalism, each terminal and non-terminal in a production can also have a *layout constraint* following it. Here, the layout constraints can be either `=` (meaning roughly the leftmost column of the annotated term must be aligned with that of the parent term), `>` (the leftmost column must be indented further), `>=` (the leftmost column *may* be indented further). Layout constraints are optional in TSLs. We will discuss this further when we formally describe Wyvern's layout-sensitive concrete syntax in the next section.

Each rule is followed by a Wyvern expression, in an indented block, that implements the rewriting logic for the associated rule. This expression should have type `Exp`, a case type built into the standard library representing Wyvern expression ASTs. The ASTs generated by named non-terminals in the rule (e.g. `child`) are bound to the corresponding variables within this logic. Here, we show how to generate an AST using general-purpose notation for `Exp` (lines 13-15) as well as the more natural *quasiquote* style (lines 11 and 18). Quasiquotes are simply the TSL associated with `Exp` and support the full Wyvern concrete syntax as well as an additional delimiter form, written with `%s`, that allows "unquoting": interpolating another AST into the one being generated. Again, interpolation is typesafe, structured and occurs at compile time.

2.5 Implementing Interpolation

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

HTML Interpolation At any point where a tag should appear, we can also interpolate 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. 3. 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 interpolation must be structured (a string cannot be interpolated directly), injection and cross-site

scripting attacks cannot occur. Safe string interpolation (which escapes any inner HTML) could be implemented using another delimiter.

CSS Interpolation 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 interpolating a Wyvern expression (lines 12-15 of Fig. 3), 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).

Interpolation within the CSS TSL The TSL for `CSS` itself has support for interpolation 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).

Interpolation within the SQLQuery TSL The TSL used for SQL queries on line 18 of Fig. 1 follows an identical pattern, allowing strings to be interpolated 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 [4]. 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 4. 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 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 to further 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. 1. 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` (not shown on the figure). 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, the general-purpose introductory form for object types and the elimination form for case types and product types all use forward referenced blocks. Fig. 5 shows all of these in use (assuming suitable definitions of casetypes `Nat` and `HTML`, not included). In the grammar,

```
(* programs *)
p → 'objtype'= ID> NEWLINE> objdecls> NEWLINE> metadatadecl> p=
p → 'casetype'= ID> NEWLINE> casedecls> NEWLINE> metadatadecl> p=
p → e=

metadatadecl → ε | 'metadata'= '='> e> NEWLINE>

(* expressions *)
e → ē=
e → ẽ['~']= NEWLINE> chars>
e → ẽ['new']= NEWLINE> d>
e → ẽ['case(' ē ')']= NEWLINE> c>

(* object definitions *)
d → ε
d → 'val'= ID> ':'> type> '='> e> NEWLINE> d=
d → 'def'= ID> '('> argsig> ')'> ':'> type> '='> e> NEWLINE> d=

(* cases *)
c → cc | cp
cc → ID= '('> ID> ')'> '=>'> e>
cc → ID= '('> ID> ')'> '=>'> e> NEWLINE> cs=
cp → '('= ID> ','> ID> ')'> '=>'> e>

(* expressions not containing forward references *)
ē → ID=
ē → 'fn'= ID> ':'> type> '=>'> ē>
ē → ē= '('> āl> ')'>
ē → '('= ē> ','> ē> ')'>
ē → 'let'= ID> ':'> type> '='> e> NEWLINE> ē=
ē → ē= '.'> ID>
ē → type= '.'> ID> '('> ē> ')'>
ē → ē= ':'> type>
ē → 'valAST'= '('> ē> ')'>
ē → type= '.'> 'metadata'>
ē → inlinelit=

āl → ε | āl_nonempty=
āl_nonempty → ē= | ē= ','> āl_nonempty>

inlinelit → chars1['‘']= | chars1['’']= | chars1['"']= | ...
inlinelit → chars2['{', '}']= | chars2['<', '>']= | chars2['[', ']']= | ...
inlinelit → numlit=

(* expressions containing exactly one forward reference *)
ẽ[fwd] → fwd=
ẽ[fwd] → 'fn'= ID> ':'> type> '=>'> ẽ[fwd]>
ẽ[fwd] → ẽ[fwd]= '('> āl> ')'>
ẽ[fwd] → '('= ē> ','> ẽ[fwd]> ')'>
ẽ[fwd] → '('= ẽ[fwd]> ','> ē> ')'>
ẽ[fwd] → 'let'= ID> ':'> type> '='> e> NEWLINE> ẽ[fwd]=
ẽ[fwd] → ē= '('> ā̃l[fwd]> ')'>
ẽ[fwd] → ẽ[fwd]= '.'> ID>
ẽ[fwd] → type= '.'> ID> '('> ẽ[fwd]> ')'>
ẽ[fwd] → ẽ[fwd]= ':'> type>
ẽ[fwd] → 'valAST'= '('> ẽ[fwd]> ')'>

ā̃l[fwd] → ẽ[fwd]= | ẽ[fwd]= ','> āl_nonempty> | ē= ','> ā̃l[fwd]>
```

Fig. 4: Concrete Syntax (a few simple productions have been omitted)

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

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, ρ, shown on the left side of Fig. 6. Forward references are internalized. 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, $\textbf{fromTS}(e)$, 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 Literal Rewriting

We will now specify a type system for the abstract syntax in Fig. 6. Conventional type systems are specified using a typechecking judgement like $\Delta; \Gamma \vdash e : \tau$, where the variable context, Γ, tracks the types of variables, and the type context, Δ, tracks types and their signatures. However, this conventional formulation does not separately consider how, when deriving this judgement, it will be considered algorithmically – will a type be provided, so that we simply need to check e against it, or do we need to synthesize a type for e? For our system, this distinction is crucial: a generic literal can only be used in the first situation.

Bidirectional type systems, as presented by Lovas and Pfenning [19], make this distinction clear by specifying the type system instead using two simultaneously defined judgements: one for expressions that can *synthesize* a type based on the surrounding context (e.g. variables and elimination forms), and another for expressions for which we know what type to *check* or *analyze* the term against (e.g. generic literals and some introductory forms). Our work builds upon this work, making the following core additions: the tye context Δ now tracks the metadata in addition to type signatures, and as we typecheck, we need to also

$$
\begin{array}{rcl}
\rho & ::= & \textbf{objtype}\ t\ \{\omega, \textbf{metadata} = e\}; \rho \\
& | & \textbf{casetype}\ t\ \{\chi, \textbf{metadata} = e\}; \rho \\
& | & e \\
\\
e & ::= & x \\
& | & \boldsymbol{\lambda} x{:}\tau.e \\
& | & e(e) \\
& | & (e, e) \\
& | & \textbf{case}(e)\{(x, y) \Rightarrow e\} \\
& | & t.C(e) \\
& | & \textbf{case}(e)\ \{c\} \\
& | & \textbf{new}\ \{d\} \\
& | & e.x \\
& | & e : \tau \\
& | & \textbf{valAST}(e) \\
& | & t.\textbf{metadata} \\
& | & \textbf{fromTS}(e) \\
& | & \lfloor body \rfloor \\
\\
c & ::= & C(x) \Rightarrow e \\
& | & c \mid c \\
\\
d & ::= & \varepsilon \\
& | & \textbf{val}\ f : \tau = e;\ d \\
& | & \textbf{def}\ m : \tau = e;\ d
\end{array}
\qquad
\begin{array}{rcl}
\hat{\rho} & ::= & \textbf{objtype}\ t\ \{\omega, \textbf{metadata} = \hat{e}\}; \hat{\rho} \\
& | & \textbf{casetype}\ t\ \{\chi, \textbf{metadata} = \hat{e}\}; \hat{\rho} \\
& | & \hat{e} \\
\\
\hat{e} & ::= & x \\
& | & \boldsymbol{\lambda} x{:}\tau.\hat{e} \\
& | & \hat{e}(\hat{e}) \\
& | & \cdots \\
& | & t.\textbf{metadata} \\
\\
\hat{c} & ::= & ... \\
\\
\hat{d} & ::= & ... \\
\\
\chi & ::= & C\ \textbf{of}\ \tau \\
& | & \chi \mid \chi \\
\\
\omega & ::= & \varepsilon \\
& | & \textbf{val}\ f : \tau;\ \omega \\
& | & \textbf{def}\ m : \tau;\ \omega \\
\\
\tau & ::= & t \\
& | & \tau \rightarrow \tau \\
& | & \tau \times \tau \\
\\
\Gamma & ::= & \emptyset \mid \Gamma, x : \tau \qquad \delta ::= - \mid \hat{e} : \tau \\
\\
\Delta & ::= & \emptyset \mid \Delta, t : \{\chi, \delta\} \mid \Delta, t : \{\omega, \delta\}
\end{array}
$$

Fig. 6: Abstract Syntax

perform literal rewriting by calling the parser associated with the type that a literal is being analyzed against, typechecking the AST it produces and ensuring that hygiene is maintained.

The judgement $\boxed{\Delta; \Gamma'; \Gamma \vdash e \Rightarrow \tau \rightsquigarrow \hat{e}}$ means that from the type context Δ, the *surrounding variable context*, Γ', and the *local variable context*, Γ, we synthesize the type τ for e and rewrite it to $\hat{e}$ (which does not contain literals or the special form $\textbf{fromTS}(e)$, which makes the surrounding context available; see below). The judgement $\boxed{\Delta; \Gamma'; \Gamma \vdash e \Leftarrow \tau \rightsquigarrow \hat{e}}$ similarly means that we check e against the type τ and the expression e is rewritten into the expression $\hat{e}$. The forms of Γ and Δ are given in Fig. 6. Note that Δ carries the type's signature as well as the rewritten form of the metadata. The rules for these judgements, as well as key rules for several auxiliary judgements that are needed in their premises, are given in Figs. 8-11.

These rules assume that a collection of built-in types are included by default at the top of programs (e.g. `Unit`, `Parser`, `Exp` already mentioned, and a few others), captured by an intial type context Δ_0. We show the concrete syntax for the two key ones in Fig. 7. The static semantics and the dynamic semantics (defined for $\hat{e}$ only) are that of a conventional functional language with functions, inductive datatypes, products and records with the addition of a few new forms. The key new dynamic semantics rules are described in Figs. 12 and 13. We will

```
objtype Parser
  def parse(ts : TokenStream) : (Exp *
    TokenStream)
  metadata = new
    val parser : Parser = new
      val parse(ts : TokenStream) : (
          Exp * TokenStream) =
        (* parser generator based
           on Adams' formalism *)
```

```
casetype Exp
    Var of ID
  | Lam of ID * Type * Exp
  | App of Exp * Exp
  ...
  | FromTS of Exp * Exp
  | Literal of TokenStream
  | Error of ErrorMessage
  metadata = (* quasiquotes *)
```

Fig. 7: Two of the built-in types included in Δ_0 (concrete syntax).

$$\frac{\Delta, t : \{\chi, -\}; \emptyset; \emptyset \vdash e \Rightarrow \tau \rightsquigarrow \hat{e} \quad \vdash \Delta, t : \{\chi, \hat{e} : \tau\} \quad \Delta, t : \{\chi, \hat{e} : \tau\} \vdash \rho : \tau' \rightsquigarrow \hat{\rho}}{\begin{array}{c}\Delta \vdash \textbf{casetype } t \ = \{\chi, \textbf{metadata} = e\}; \rho : \tau' \rightsquigarrow \\ \textbf{casetype } t \ = \{\chi, \textbf{metadata} = \hat{e}\}; \hat{\rho}\end{array}} \ \textit{RT-CT}$$

$$\frac{\Delta \vdash \tau}{\Delta \vdash C \textbf{ of } \tau \ \mathsf{ok}} \ \textit{C-decl} \qquad \frac{\Delta \vdash \chi_1 \ \mathsf{ok} \quad \Delta \vdash \chi_2 \ \mathsf{ok} \quad \mathrm{dom}(\chi_1) \cap \mathrm{dom}(\chi_2) = \emptyset}{\Delta \vdash \chi_1 \mid \chi_2 \ \mathsf{ok}} \ \textit{C-decls}$$

$$\frac{\Delta, t : \{\omega, -\}; \emptyset; \emptyset \vdash e \Rightarrow \tau \rightsquigarrow \hat{e} \quad \vdash \Delta, t : \{\omega, \hat{e} : \tau\} \quad \Delta, t : \{\omega, \hat{e} : \tau\} \vdash \rho : \tau' \rightsquigarrow \hat{\rho}}{\begin{array}{c}\Delta \vdash \textbf{objtype } t \ = \{\omega, \textbf{metadata} = e\}; \rho : \tau' \rightsquigarrow \\ \textbf{objtype } t \ = \{\omega, \textbf{metadata} = \hat{e}\}; \hat{\rho}\end{array}} \ \textit{RT-OT}$$

$$\frac{\Delta \vdash \tau \quad f \notin \omega \quad \Delta \vdash \omega \ \mathsf{ok}}{\Delta \vdash \textbf{val } f : \tau; \omega \ \mathsf{ok}} \ \textit{O-v} \qquad \frac{\Delta \vdash \tau \quad m \notin \omega \quad \Delta \vdash \omega \ \mathsf{ok}}{\Delta \vdash \textbf{def } m : \tau; \omega \ \mathsf{ok}} \ \textit{O-d} \qquad \frac{}{\Delta \vdash \epsilon \ \mathsf{ok}} \ \textit{O-e}$$

$$\frac{\Delta; \emptyset; \emptyset \vdash e \Rightarrow \tau \rightsquigarrow \hat{e}}{\Delta \vdash e : \tau \rightsquigarrow \hat{e}} \ \textit{RT-e}$$

Fig. 8: Statics for Programs

now describe how some of the novel rules that support TSLs work below. We refer the reader to [19] and texts on type systems, e.g. [14, 25], for the remainder.

4.1 Defining a TSL Manually

In the example in Fig. 3, we showed a TSL being defined using a parser generator based on Adams' formalism. A parser generator is itself merely a TSL for a parser, and a parser is the fundamental construct that becomes associated with a type to form a TSL. The signature for the built-in type `Parser` is shown in Fig. 7. It is an object type with a `parse` function taking in a `TokenStream` and producing an AST of a Wyvern expression, which is of type `Exp`. This built-in type is shown also in Fig. 7. Note that there is a form for each form in the abstract syntax, e, as well as an `Error` form for indicating error messages (in the theory, nothing is done with these messages). As previously mentioned, quasiquotes are merely a TSL that allows one to construct the abstract syntax, represented as this case type, using concrete syntax, with the addition of an unquote mechanism.

The `parse` function for a type t is called when checking a literal form against that type. This is seen in the key rule of our statics: *T-lit*, in Fig. 9. The premises of these rules operate as follows:

1. This rule uses some built-in types. We first ensure they are available.

$$\frac{\Delta;\Gamma';\Gamma \vdash e \Rightarrow \tau \rightsquigarrow \hat{e}}{\Delta;\Gamma';\Gamma \vdash e \Leftarrow \tau \rightsquigarrow \hat{e}}\ \textit{S2C} \qquad \frac{x:\tau \in \Gamma}{\Delta;\Gamma';\Gamma \vdash x \Rightarrow \tau}\ \textit{T-var}$$

$$\frac{\Delta \vdash \tau \quad \Delta;\Gamma';\Gamma, x:\tau \vdash e \Leftarrow \tau_1 \rightsquigarrow \hat{e}}{\Delta;\Gamma';\Gamma \vdash \boldsymbol{\lambda} x{:}\tau.e \Leftarrow \tau \to \tau_1 \rightsquigarrow \boldsymbol{\lambda} x{:}\tau.\hat{e}}\ \textit{T-abs}$$

$$\frac{\Delta;\Gamma';\Gamma \vdash e \Rightarrow \tau_1 \to \tau_2 \rightsquigarrow \hat{e} \quad \Delta;\Gamma';\Gamma \vdash_{\Gamma'} e_1 \Leftarrow \tau_1 \rightsquigarrow \hat{e}_1}{\Delta;\Gamma';\Gamma \vdash e(e_1) \Rightarrow \tau_2 \rightsquigarrow \hat{e}(\hat{e}_1)}\ \textit{T-appl}$$

$$\frac{t:\{\chi,\delta\} \in \Delta \quad C\ \mathbf{of}\ \tau' \in \chi \quad \Delta;\Gamma';\Gamma \vdash e \Leftarrow \tau' \rightsquigarrow \hat{e}}{\Delta;\Gamma';\Gamma \vdash t.C(e) \Rightarrow t \rightsquigarrow t.C(\hat{e})}\ \textit{T-introcase}$$

$$\frac{\Delta;\Gamma';\Gamma \vdash e \Rightarrow t \rightsquigarrow \hat{e} \quad t:\{\chi,\delta\} \in \Delta \quad \Delta;\Gamma';\Gamma \vdash c:\chi \Rightarrow \tau' \rightsquigarrow \hat{c}}{\Delta;\Gamma';\Gamma \vdash \mathbf{case}\ (e)\ \{c\} \Rightarrow \tau' \rightsquigarrow \mathbf{case}\ (\hat{e})\ \{\hat{c}\}}\ \textit{T-elimcase}$$

$$\frac{\Delta;\Gamma';\Gamma, x:\tau \vdash e \Rightarrow \tau' \rightsquigarrow \hat{e}}{\Delta;\Gamma';\Gamma \vdash C(x) \Rightarrow e : C\ \mathbf{of}\ \tau \Rightarrow \tau' \rightsquigarrow C(x) \Rightarrow \hat{e}}\ \textit{T-casehelper1}$$

$$\frac{\Delta;\Gamma';\Gamma \vdash c_1:\chi_1 \Rightarrow \tau' \rightsquigarrow \hat{c}_1 \quad \Delta;\Gamma';\Gamma \vdash c_2:\chi_2 \Rightarrow \tau' \rightsquigarrow \hat{c}_2}{\Delta;\Gamma';\Gamma \vdash c_1 \mid c_2:\chi_1 \mid \chi_2 \Rightarrow \tau' \rightsquigarrow \hat{c}_1 \mid \hat{c}_2}\ \textit{T-casehelper2}$$

$$\frac{\Delta;\Gamma';\Gamma \vdash e_1 \Leftarrow \tau_1 \rightsquigarrow \hat{e}_1 \quad \Delta;\Gamma';\Gamma \vdash e_2 \Leftarrow \tau_2 \rightsquigarrow \hat{e}_2}{\Delta;\Gamma';\Gamma \vdash (e_1,e_2) \Leftarrow \tau_1 \times \tau_2 \rightsquigarrow (\hat{e}_1,\hat{e}_2)}\ \textit{T-prod-intro}$$

$$\frac{\Delta;\Gamma';\Gamma \vdash e \Rightarrow \tau_1 \times \tau_2 \rightsquigarrow \hat{e} \quad \Delta;\Gamma';\Gamma, x:\tau_1, y:\tau_2 \vdash e' \Rightarrow \tau \rightsquigarrow \hat{e}'}{\Delta;\Gamma';\Gamma \vdash \mathbf{case}(e)\{(x,y) \Rightarrow e'\} \Rightarrow \tau \rightsquigarrow \mathbf{case}(\hat{e})\{(x,y) \Rightarrow \hat{e}'\}}\ \textit{T-prod-elim}$$

$$\frac{t:\{\omega,\delta\} \in \Delta \quad \Delta;\Gamma';\Gamma \vdash_t d \Leftarrow \omega \rightsquigarrow \hat{d} \quad t \neq \mathit{TokenStream}}{\Delta;\Gamma';\Gamma \vdash \mathbf{new}\ \{d\} \Leftarrow t \rightsquigarrow \mathbf{new}\ \{\hat{d}\}}\ \textit{T-obj-intro}$$

$$\frac{\Delta \vdash \tau \quad \Delta;\Gamma';\Gamma \vdash e \Leftarrow \tau \rightsquigarrow \hat{e} \quad \Delta;\Gamma';\Gamma \vdash_t d \Leftarrow \omega \rightsquigarrow \hat{d}}{\Delta;\Gamma';\Gamma \vdash_t \mathbf{val}\ f:\tau = e; d \Leftarrow \mathbf{val}\ f:\tau; \omega \rightsquigarrow \mathbf{val}\ f:\tau = \hat{e}; \hat{d}}\ \textit{DT-val}$$

$$\frac{\Delta \vdash \tau \quad \Delta;\Gamma';\Gamma \vdash e \Leftarrow t \to \tau \rightsquigarrow \hat{e} \quad \Delta;\Gamma';\Gamma \vdash_t d \Leftarrow \omega \rightsquigarrow \hat{d}}{\Delta;\Gamma';\Gamma \vdash_t \mathbf{def}\ m:\tau = e; d \Leftarrow \mathbf{def}\ m:\tau; \omega \rightsquigarrow \mathbf{def}\ m:\tau = \hat{e}; \hat{d}}\ \textit{DT-def}$$

$$\frac{}{\Delta;\Gamma';\Gamma \vdash_t \epsilon \Leftarrow \epsilon \rightsquigarrow \epsilon}\ \textit{DT-emp}$$

$$\frac{\Delta;\Gamma';\Gamma \vdash e \Rightarrow t \rightsquigarrow \hat{e} \quad t:\{\omega,\hat{e}_0:\tau\} \in \Delta \quad \mathbf{val}\ f:\tau' \in \omega}{\Delta;\Gamma';\Gamma \vdash e.f \Rightarrow \tau' \rightsquigarrow \hat{e}.f}\ \textit{T-field}$$

$$\frac{\Delta;\Gamma';\Gamma \vdash e \Rightarrow t \rightsquigarrow \hat{e} \quad t:\{\omega,\delta\} \in \Delta \quad \mathbf{def}\ m:\tau' \in \omega}{\Delta;\Gamma';\Gamma \vdash e.m \Rightarrow \tau' \rightsquigarrow \hat{e}.m}\ \textit{T-def}$$

Fig. 9: Statics for e (continued below; see Appendix for well-formedness rules)

$$\frac{\Delta \vdash \tau \quad \Delta; \Gamma' ; \Gamma \vdash e \Leftarrow \tau \rightsquigarrow \hat{e}}{\Delta; \Gamma' ; \Gamma \vdash e : \tau \Rightarrow \tau \rightsquigarrow \hat{e} : \tau} \; \textit{T-ascribe}$$

$$\frac{\vdash \Delta_0, \Delta \quad \Delta_0, \Delta; \Gamma' ; \Gamma \vdash e \Rightarrow \tau \rightsquigarrow \hat{e}}{\Delta_0, \Delta; \Gamma' ; \Gamma \vdash \mathbf{valAST}(e) \Rightarrow \mathit{Exp} \rightsquigarrow \mathbf{valAST}(\hat{e})} \; \textit{T-valAST}$$

$$\frac{t : \{\_, \hat{e}_0 : \tau\} \in \Delta}{\Delta; \Gamma' ; \Gamma \vdash t.\mathbf{metadata} \Rightarrow \tau \rightsquigarrow t.\mathbf{metadata}} \; \textit{T-metadata}$$

$$\frac{\Delta; \emptyset; \Gamma', \Gamma \vdash e \Leftarrow \tau \rightsquigarrow \hat{e}}{\Delta; \Gamma' ; \Gamma \vdash \mathbf{fromTS}(e) \Leftarrow \tau \rightsquigarrow \hat{e}} \; \textit{T-fromTS}$$

$$\frac{\begin{array}{c}\vdash \Delta_0, \Delta \quad \Delta_0, \Delta; \emptyset; \emptyset \vdash t.\mathbf{metadata}.\mathit{parser} \Leftarrow \mathit{Parser} \rightsquigarrow \hat{e}_p \quad \texttt{TS(}\lfloor \mathit{body} \rfloor\texttt{) is } \hat{e}_{ts} \\ \hat{e}_p.\mathit{parse}(\hat{e}_{ts}) \Downarrow_{\Delta_0, \Delta} (\hat{e}', \hat{e}'_{ts}) \quad e \triangleleft \hat{e}' \quad \Delta_0, \Delta; \Gamma', \Gamma; \emptyset \vdash e \Leftarrow t \rightsquigarrow \hat{e} \quad \hat{e}'_{ts}\ \texttt{empty}\end{array}}{\Delta_0, \Delta; \Gamma' ; \Gamma \vdash \lfloor \mathit{body} \rfloor \Leftarrow t \rightsquigarrow \hat{e}} \; \textit{T-lit}$$

Fig. 10: Statics for e, continued

$$\frac{t : \{\omega, \delta\} \in \Delta \quad \Delta; \Gamma \vdash_t \hat{d} : \omega}{\Delta; \Gamma \vdash \mathbf{new}\ \{\hat{d}\} : t} \; \textit{T-new-hat}$$

Fig. 11: Statics for $\hat{e}$. All rules are the unidirectional version of the corresponding rule for e with a combined variable context. **new** is unrestricted. See Lemma 1.

2. A well-typed, rewritten parser object is extracted from the type's metadata. This is the step where the parser generator rewrites a grammar to a parse method, recursively using the TSL mechanism itself.
3. A tokenstream, of type `TokenStream`, is generated from the body of the literal. This type is an object that that allows the reading of tokens, as well as an additional method discussed in the next section for parsing the stream as a Wyvern expression.
4. The `parse` method is called with this extracted tokenstream to produce a syntax tree and a remaining tokenstream.
5. The syntax tree, $\hat{e}'$ is *dereified* into its corresponding term, e (the hat is gone because the generated syntax tree might itself use TSLs). This is the only way terms of the form $\mathbf{fromTS}(e)$ can be generated (see below).
6. The dereified term is then recursively typechecked against the same type and rewritten, consistent with the semantics of TSLs as we have been describing them – they must produce a term of the type they are being checked against. It is checked under the empty local context to ensure hygiene (below).
7. The TSL must consume the entire token stream, so this is checked.

4.2 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

$$\frac{\texttt{Var}(\hat{e}_{id})\ \texttt{is}\ x}{x \triangleleft Exp.Var(\hat{e}_{id})}\ \textit{DR-Var}$$

$$\frac{\texttt{Var}(\hat{e}_{id})\ \texttt{is}\ x \quad \tau \triangleleft \hat{e}_{ty} \quad e \triangleleft \hat{e}}{\boldsymbol{\lambda} x{:}\tau.e \triangleleft Exp.Lam(\hat{e}_{id}, \hat{e}_{ty}, \hat{e})}\ \textit{DR-Lam}$$

$$\frac{e_1 \triangleleft \hat{e}_1 \quad e_2 \triangleleft \hat{e}_2}{e_1(e_2) \triangleleft Exp.App(\hat{e}_1, \hat{e}_2)}\ \textit{DExp-App}$$

$$\frac{\texttt{Literal}(\hat{e}_{ts})\ \texttt{is}\ \lfloor body \rfloor}{\lfloor body \rfloor \triangleleft Exp.Literal(\hat{e}_{ts})}\ \textit{DExp-Lit}$$

$$\frac{\texttt{ParseConc}(\hat{e}_{ts}, \hat{e}_{tok})\ \texttt{is}\ e}{\begin{array}{l}\textbf{fromTS}(e) \triangleleft \\ Exp.FromTS(\hat{e}_{ts}, \hat{e}_{tok})\end{array}}\ \textit{DR-FromTS}$$

$$\frac{\texttt{Var}(\hat{e}_{id})\ \texttt{is}\ t}{t \triangleleft Ty.Var(\hat{e}_{id})}\ \textit{DTy-Var}$$

$$\frac{\tau_1 \triangleleft \hat{e}_1 \quad \tau_2 \triangleleft \hat{e}_2}{\tau_1 \to \tau_2 \triangleleft Ty.Arrow(\hat{e}_1, \hat{e}_2)}\ \textit{DTy-Arr}$$

Fig. 12: Dereification Rules (selected)

$$\frac{\texttt{ID}(x)\ \texttt{is}\ \hat{e}_{id}}{x \triangleright Exp.Var(\hat{e}_{id})}\ \textit{R-Var}$$

$$\frac{\texttt{ID}(x)\ \texttt{is}\ \hat{e}_{id} \quad \tau \triangleright \hat{e}_{ty} \quad \hat{e} \triangleright \hat{e}'}{\boldsymbol{\lambda} x{:}\tau.\hat{e} \triangleright Exp.Lam(\hat{e}_{id}, \hat{e}_{ty}, \hat{e}')}\ \textit{R-Lam}$$

$$\frac{\hat{e}_1 \triangleright \hat{e}_1' \quad \hat{e}_2 \triangleright \hat{e}_2'}{\hat{e}_1(\hat{e}_2) \triangleright Exp.App(\hat{e}_1', \hat{e}_2')}\ \textit{R-App}$$

$$\frac{\texttt{ID}(t)\ \texttt{is}\ \hat{e}_{id}}{t \triangleright Ty.Var(\hat{e}_{id})}\ \textit{RTy-Var}$$

$$\frac{\tau_1 \triangleright \hat{e}_1 \quad \tau_2 \triangleright \hat{e}_2}{\tau_1 \to \tau_2 \triangleright Ty.Arrow(\hat{e}_1, \hat{e}_2)}\ \textit{RTy-Arr}$$

Fig. 13: Reification Rules (selected)

$$\frac{t : \{\_, \hat{e} : \tau\} \in \Delta}{t.\textbf{metadata} \underset{\Delta}{\longmapsto} \hat{e}}\ \textit{Dyn-Meta}$$

$$\frac{\hat{e} \underset{\Delta}{\longmapsto} \hat{e}'}{\textbf{valAST}(\hat{e}) \underset{\Delta}{\longmapsto} \textbf{valAST}(\hat{e}')}\ \textit{Dyn-valAST1} \qquad \frac{\hat{e}\ \texttt{val} \quad \hat{e} \triangleright \hat{e}'}{\textbf{valAST}(\hat{e}) \underset{\Delta}{\longmapsto} \hat{e}'}\ \textit{Dyn-valAST2}$$

Fig. 14: Dynamic Semantics **[TODO: full (de)reification/opsem]**

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. Strict hygiene would simply reject all open terms, but this would prevent even nested Wyvern expressions which the user provided from referring to local variables.

The solution to being able to capture variables in portions of the tokenstream that are parsed as Wyvern only is to add a new term to the abstract syntax that has no corresponding form in the concrete syntax: **fromTS**(e). This means: "this is a term that was parsed from the user's tokenstream". It can be generated by calling `ts.as_wyv_exp(tok)`, which returns a the remaining tokenstream as well as the value `Exp.FromTS(ts, tok)`. When we dereify this term, we turn this into the

form **fromTS**(e), where e is the result of parsing the tokenstream as a Wyvern expression until an unexpected token `tok` appears.

When we attempt to typecheck this form, which will be starting from an empty local variable context by moving all the available variables into the surrounding variable context (in the *T-Lit* rule, above), we add in the bindings available in the surrounding variable context (to any that were introduced by the TSL, such as the TSL for `Parser` does with named non-terminals). In other words, variables in the surrounding variable context can only be used within a term of the form **fromTS**(e). These variables are precisely those that only the user can know exist, but not the extension.

For this mechanism to truly ensure hygiene, one must not be able to sidestep it by generating a tokenstream manually: expressions from a tokenstream must have actually come from the use site. This is ensured by preventing users from checking `new` against `TokenStream` in the statics.

A second facet of hygiene is being able to refer to local variables available within the parser itself, such as local helper functions, for convenience. This can be done using the primitive **valAST**(e). The semantics for this, shown in Fig. 13, first evaluate e to a value, then *reify* this value to an AST. 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 essentially dual, as seen in Figs. 11 and 12.

4.3 Safety

The semantics we have defined constitute a type safe language.

We begin with a lemma that shows that the statics for e and $\hat{e}$ are consistent. This makes us sure that the splitting of variable contexts to maintain hygiene was done correctly (because they can be brought back together at the end).

Lemma 1 (Forward Consistency).

1. If $\vdash \Delta$ *and* $\Delta \vdash \Gamma'$ *and* $\Delta \vdash \Gamma$ *and* $\Delta; \Gamma'; \Gamma \vdash e \Leftarrow \tau \rightsquigarrow \hat{e}$ *then* $\Delta; \Gamma', \Gamma \vdash \hat{e} : \tau$.
2. If $\vdash \Delta$ *and* $\Delta \vdash \Gamma'$ *and* $\Delta \vdash \Gamma$ *and* $\Delta; \Gamma'; \Gamma \vdash e \Rightarrow \tau \rightsquigarrow \hat{e}$ *then* $\Delta; \Gamma', \Gamma \vdash \hat{e} : \tau$.

Proof. Forward consistency is easily seen by observing that for each form shared by both e and $\hat{e}$, the bidirectional system simply rewrites to the corresponding form of $\hat{e}$ recursively. Thus, these cases are direct applications of the IH. For the literal form, we can apply the IH to arrive at the fact that $\Delta_0, \Delta; \Gamma', \Gamma, \emptyset \vdash \hat{e} : t$ which by congruence (removing the empty context at the end) is what we wish to show. Similarly, for the form **fromTS**(e) we have by the IH that $\Delta; \emptyset, \Gamma', \Gamma \vdash \hat{e} : \tau$ which again implies what we wish to show by simple congruence of contexts.

We then need to show type safety of $\hat{e}$. Because it doesn't contain any non-standard terms other than **valAST** and t.**metadata**, both of which have straightforward semantics (Fig. 13), this follows by the standard progress and preservation techniques. The only tricky case is *Dyn-valAST2*, which requires the following straightforward lemma about the reification rules in Fig. 12, as well as standard structural properties for the contexts (weakening; not shown).

Lemma 2 (Reification). *If* $\hat{e} \triangleright \hat{e}'$ *then* $\Delta_0; \emptyset \vdash \hat{e}' : Exp$.

Lemma 3 (Preservation). *If* $\vdash \Delta$ *and* $\Delta; \emptyset \vdash \hat{e} : \tau$ *and* $\hat{e} \underset{\Delta}{\longmapsto} \hat{e}'$ *then* $\Delta; \emptyset \vdash \hat{e}' : \tau$.

Lemma 4 (Progress). *If* $\vdash \Delta$ *and* $\Delta; \emptyset \vdash \hat{e} : \tau$ *then either* $\hat{e}$ `val` *or* $\hat{e} \underset{\Delta}{\longmapsto} \hat{e}'$.

These lemmas and associated judgements can be lifted to the level of programs by applying them to the top-level expression the program contains (simple, not shown). As a result, we have type safety: well-typed programs cannot "get stuck".

Theorem 1 (Type Safety). *If* $\Delta_0 \vdash \rho : \tau \rightsquigarrow \hat{\rho}$ *then* $\Delta_0 \vdash \hat{\rho} : \tau$ *and either* $\hat{\rho}$ `val` *or* $\hat{\rho} \underset{\Delta_0}{\longmapsto} \hat{\rho}'$ *such that* $\Delta_0 \vdash \hat{\rho}' : \tau$.

4.4 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, or it might generate a term that contains itself. 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 $\hat{e}$ (which we do not actually need to do in practice by Lemma 1). Termination of parsers and parser generators has previously been studied (e.g. [17]) 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 [**?**]), 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

An important question when introducing a new approach is how it would change the existing solutions and at what places it could be used. To answer these questions we performed a code analysis and tried to identify potential uses of the TSLs in the existing Java code. Our analysis is limited in scope and focused only on the class constructors. We are interested in class constructors because they are the programmatic constructs that potentially could be equipped with Wyvern types. For example, a Java class constructor such as

```
1  Path(String path) {...}
```

which takes in a single `String` and makes sure that it is of a specific format, could be equipped with a Wyvern type `Path` that would check for the format of the string. We examined this type of Java constructors in the first part of our analysis.

Further, having the pool of Java constructors we wanted to see in how many of them a TSL could be used. A place where we believe a TSL could be used is a `String` argument that has a specific format, for instance, constructors such as:

```
1  FileUpdatedEvent(Object source, String path) {...}
```

Here, the second argument `path`, which is of type `String`, could be represented using a Wyvern type `Path` that would guarantee that the passed in argument is of the required format.

Methodology To perform our analysis, we used a recent version (20130901r) of 107 Java projects in the Qualitas Corpus [29] and searched for the two types of constructors described above. We used command line tools, such as `grep` and `sed`, and editor features such as search and substitution. In a semi-manual procedure, we scanned though the Java code and picked out class constructors. After that, we chose constructors that take at least one `String` as an argument, and looking at the names of the constructors and their arguments, we inferred the intended use of the classes associated with them.

Constructors	**Number**	**% of Total**
Total analyzed	124,873	100
Have a String argument	30,161	24
Could be equipped with a TSL	603	0.5
Could use a TSL	19,288	15

Table 1: Summary of the Analyses Results

Type of String	**Number**	**% of Total**
Name	14,307	74.2
ID	1,335	6.9
Directory path	823	4.3
Pattern	495	2.6
URL/URI	396	2.0
Other (zip code, password, query, HTML/XML, IP address, version, etc.)	1,932	10.0
Total:	**19,288**	**100.0**

Table 2: Types of `String` Arguments in Java Constructors

Results Our findings are summarized in Figure 1 and 2. For the first part of our analysis, looking through the Java constructors, we found that there is 0.5% (603 out of 124,873 examined constructors) which could be equipped with a TSL. Those constructors were used for classes that represent URLs and URIs, identification numbers, versions, directory paths, and various types of names (e.g., user name, database name, column name, etc.).

For the second part of our analysis, we found that there is 15% (19,288 out of 124,873 constructors) of constructors that could use a TSL. More details on

the kinds of `String` arguments that are passed into constructors can be found in Table 2. The "Name" category refers to the name of a file, a user, a class, etc. that do not have to be unique; the "ID" category comprises process IDs, user IDs, column or row IDs, etc. that must have the uniqueness property; 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.

Hence, our empirical study has shown that there is significant portion of Java constructors that have a potential of taking advantage of TSLs. It is important to keep in mind that our analysis was narrow: it focused exclusively on the constructors and thus forwent many other types of programming constructs, such as methods, variable assignments, etc., that could possibly also benefit from our approach.

6 Implementation

As of this writing, our implementation of the techniques described herein is ongoing based on a fork of the public Wyvern implementation. The Wyvern implementation is written in Java, based around a custom recursive-descent parser, with self-hosting left as future work. Our continued use of the Wyvern custom parser preceded our awareness of Adams' formalism (which does not have a public implementation currently). Thus, it is implemented with a stateful lexer as in Python, producing `INDENT` and `DEDENT` tokens. The token stream produced by the lexer is then passed into the Wyvern parser. When a language transition occurs, the Wyvern core parser extracts a substream from the current token stream, using either `INDENT` and `DEDENT` or any of the TSL delimiters to indicate where the substream should begin or end. This substream is then passed to the extension parser as an argument. By subdividing the token stream, the parsers can avoid complicated issues with delegation of responsibility caused by a single shared stream. We anticipate shifting to a more elegant system based on what we have specified here by the time the paper is presented. Some extension parsers are added though the interpreter's Java interoperability support.

In order to invoke the correct extension, we combine the typechecking and parsing stages of the compiler, so that typechecking happens incrementally as semantically distinct portions of the source are parsed. Once the first stage of parsing is complete and all Wyvern expressions are known, the Wyvern constructs are typechecked. Then, types for TSL blocks are inferred from the local type context, and the associated parsers are invoked in the next stage on the substreams inside the TSL blocks. This process then continued recursively, as TSL blocks can contain Wyvern code that contains TSLs and etc., until all expressions have been parsed and typechecked. This current implementation is as described in this paper.

7 Discussion

Safe TSL Composition Our primary contribution is a strategy, where nesting of TSLs occurs by briefly entering the 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. 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 [12, 16].

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. We could require that only the *declared* type's grammar is used (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. Wyvern does not currently support subtyping, so we leave this as future work.

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.

8 Related Work

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 [20]). 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 needing to invoke 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) [28] provides a macro system based on a meta-object protocol, and Backstage Java [24], Template Haskell [27] and others employ compile-time meta-programming. 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 [9] is a preprocessor for OCaml that offers the developer the ability to extend the concrete syntax of the language via the use of parsers and extensible grammars. SugarJ [11] takes a library-centric approach which supports syntactic extension of the Java language by adding libraries. In Wyvern, 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 [6]. 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. We believe that this is a more simple and flexible methodology. C# expression trees [1] 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 [7] 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 [26] 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 [15], Ensō [8] and others [16, 30], 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 [18] 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.

In addition, there is an ongoing work on projectional editors (e.g., [2, 10]) that use special graphical user interface to allow the developer to implicitly mark where the extensions are placed in the code, essential specifying directly 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.

There is a relation between recent work on Active Code Completion and our approach in that the Active Code Completion work associates code completion palettes with types [23] as well. Such palettes could be used for defining a TSL syntax for types. However, that syntax is immediately translated to Java syntax at edit time, while this work integrates with the core parsing facilities of the language.

9 Conclusion

In this paper, we described how extensible parsing in Wyvern makes for a solid platform to support whitespace-delimited, type-directed embedded DSLs or *Type-Specific Languages (TSLs)* for short. In the future, we aim to implement a wide variety of TSLs in Wyvern tweaking our approach and implementation thereof to provide a comprehensive example of supporting multiple interacting TSLs in a safe and easy-to-use manner.

References

1. Expression Trees (C# and Visual Basic). `http://msdn.microsoft.com/en-us/library/bb397951.aspx`.
2. JetBrains MPS – Meta Programming System. `http://www.jetbrains.com/mps/`.
3. OWASP Top 10 2013. `https://www.owasp.org/index.php/Top_10_2013-Top_10`, 2013.
4. M. D. Adams. Principled parsing for indentation-sensitive languages: Revisiting landin's offside rule. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 511–522, New York, NY, USA, 2013. ACM.

5. M. Bravenboer, E. Dolstra, and E. Visser. Preventing injection attacks with syntax embeddings. In *Proceedings of the 6th International Conference on Generative Programming and Component Engineering*, GPCE '07, pages 3–12, New York, NY, USA, 2007. ACM.
6. 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.
7. T. Clark, P. Sammut, and J. S. Willans. Beyond annotations: A proposal for extensible java (XJ). In *Source Code Analysis and Manipulation*, 2008.
8. W. R. Cook, A. Loh, and T. van der Storm. Ensō: A self-describing DSL workbench. `http://enso-lang.org/`.
9. D. de Rauglaudre. *Camlp4 - Reference Manual*, 2003.
10. L. Diekmann and L. Tratt. Parsing composed grammars with language boxes. In *Workshop on Scalable Language Specification*, 2013.
11. S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. SugarJ: library-based language extensibility. In *Object-Oriented Programming Systems, Languages, and Applications*, 2011.
12. S. Erdweg and F. Rieger. A framework for extensible languages. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, GPCE '13, pages 3–12, New York, NY, USA, 2013. ACM.
13. 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.
14. R. Harper. *Practical foundations for programming languages*. Cambridge University Press, 2012.
15. 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.
16. H. Krahn, B. Rumpe, and S. Völkel. Monticore: Modular development of textual domain specific languages. In *Objects, Components, Models and Patterns*, 2008.
17. L. Krishnan and E. V. Wyk. Termination analysis for higher-order attribute grammars. In *SLE*, pages 44–63, 2012.
18. 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.
19. W. Lovas and F. Pfenning. A bidirectional refinement type system for lf. In *Electronic Notes in Theoretical Computer Science, 196:113–128, January 2008. [NPP07] [Pfe92] [Pfe93] [Pfe01] Aleksandar Nanevski, Frank Pfenning, and Brigitte*, 2008.
20. E. Möller. SRFI-49: Indentation-sensitive syntax. `http://srfi.schemers.org/srfi-49/srfi-49.html`, 2005.
21. 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.
22. 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.
23. C. Omar, Y. Yoon, T. D. LaToza, and B. A. Myers. Active code completion. In *International Conference on Software Engineering*, 2012.

24. Z. Palmer and S. F. Smith. Backstage Java: Making a Difference in Metaprogramming. In *Object-Oriented Programming Systems, Languages, and Applications*, 2011.
25. B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
26. 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.
27. T. Sheard and S. Jones. Template meta-programming for haskell. *ACM SIGPLAN Notices*, 37(12):60–75, 2002.
28. M. Tatsubori, S. Chiba, M.-O. Killijian, and K. Itano. OpenJava: A Class-based Macro System for Java. In *Reflection and Software Engineering*, 2000.
29. 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)*, pages 336–345, Dec. 2010.
30. M. G. J. van den Brand. *Pregmatic: A Generator for Incremental Programming Environments*. PhD thesis, Katholieke Universiteit Nijmegen, 1992.

A Appendix

$$\frac{}{\vdash \emptyset}\ \textit{T-D-E} \qquad \frac{t \notin \mathit{dom}(\Delta) \quad \Delta, t : \{\chi, -\} \vdash \chi \quad \Delta, t : \{\chi, -\} \vdash \delta}{\vdash \Delta, t : \{\chi, \delta\}}\ \textit{T-D-C}$$

$$\frac{t \notin \mathrm{dom}(\Delta) \quad \Delta, t : \{\omega, -\} \vdash \omega \quad \Delta, t : \{\omega, -\} \vdash \delta}{\vdash \Delta, t : \{\omega, \delta\}}\ \textit{T-D-O}$$

$$\frac{}{\Delta \vdash -}\ \textit{T-d-e} \qquad \frac{\Delta, \emptyset \vdash \hat{e} : \tau}{\Delta \vdash \hat{e} : \tau}\ \textit{T-d-m} \qquad \frac{}{\Delta \vdash \emptyset}\ \textit{T-DC-e} \qquad \frac{\Delta \vdash \Gamma \quad \Delta \vdash \tau}{\Delta \vdash \Gamma, x : \tau}\ \textit{T-DC-t}$$

$$\frac{t \in \mathrm{dom}(\Delta)}{\Delta \vdash t}\ \textit{T-T-v} \qquad \frac{\Delta \vdash \tau_1 \quad \Delta \vdash \tau_2}{\Delta \vdash \tau_1 \to \tau_2}\ \textit{T-T-a} \qquad \frac{\Delta \vdash \tau_1 \quad \Delta \vdash \tau_2}{\Delta \vdash \tau_1 \times \tau_2}\ \textit{T-T-p}$$

Fig. 15: Context and type well-formedness rules